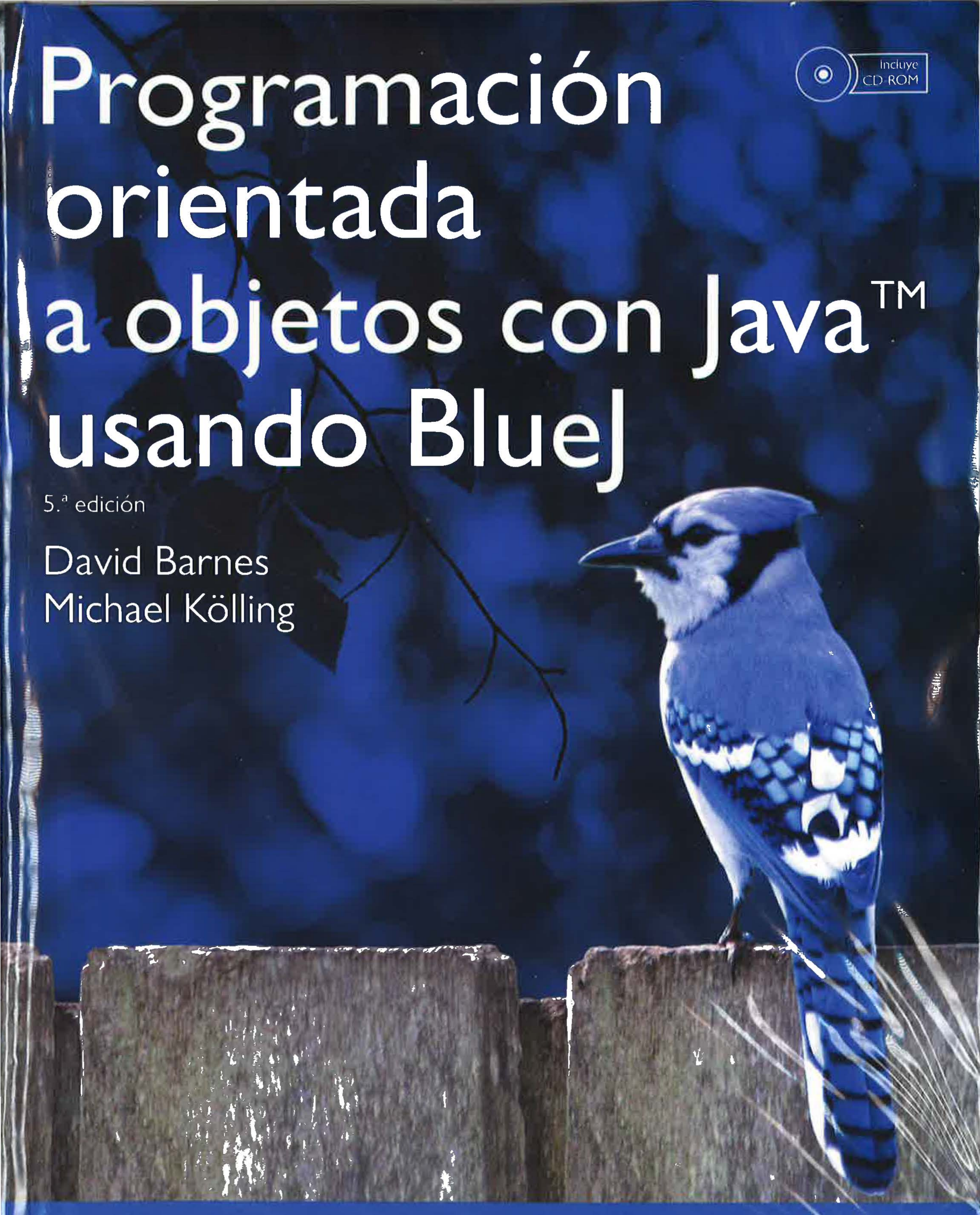


Programación orientada a objetos con Java™ usando BlueJ
5.ª edición
David Barnes
Michael Kölling
Incluye CD ROM
ALWAYS LEARNING
PEARSON

ACCESO ONLINE

para *Programación orientada a objetos con Java™ usando BlueJ*, 5.ª edición

Gracias por adquirir una nueva copia de ***Programación orientada a objetos con Java™ usando BlueJ***. Este libro de texto incluye seis meses de acceso gratuito a las *VideoNotes* del libro. Esta suscripción le proporciona acceso completo a las siguientes áreas de soporte para el estudiante:

- *VideoNotes* es una nueva herramienta visual de Pearson diseñada para enseñar a los estudiantes los conceptos y técnicas principales de la programación. Estos cortos vídeos muestran paso a paso cómo resolver problemas que van desde el diseño a la codificación. *VideoNotes* le permite planificar su formación a su propio ritmo, proporcionando una navegación fácil que incluye la posibilidad de seleccionar, reproducir, rebobinar, avanzar y detenerse dentro de cada ejercicio *VideoNote*.



ALWAYS LEARNING PEARSON

Utilice una moneda para raspar la banda y ver su código de acceso de estudiante. No utilice un cuchillo ni cualquier otro objeto cortante, ya que puede dañar el código.

Para acceder a las VideoNotes de ***Programación orientada a objetos con Java™ usando BlueJ***, la primera vez que lo haga, tendrá que registrarse en línea utilizando una computadora que disponga de una conexión a Internet y de un explorador web. El proceso solo le llevará unos pocos minutos y solo tendrá que llevarlo a cabo una vez.

1. Acceda a http://www.pearsonhighered.com/barnes_kolling/
2. Haga clic en **VideoNotes**.
3. Haga clic en el botón **Register**.
4. En la página de registro, introduzca el código de acceso de estudiante* que ha encontrado después de rascar el panel anterior. No escriba los guiones. Puede utilizar letras minúsculas o mayúsculas.
5. Siga las instrucciones que se muestran en pantalla. Si en cualquier momento mientras se está registrando necesita ayuda, simplemente haga clic en el icono **Need Help?**
6. Una vez que haya confirmado su nombre de usuario y su contraseña, podrá comenzar a utilizar *VideoNotes* para *Programación orientada a objetos con Java™ usando BlueJ*.

Cómo acceder una vez registrado

Solo es necesario registrarse una vez para acceder a *VideoNotes*. Después, puede iniciar una sesión en cualquier momento accediendo a *http://www.pearsonhighered.com/barnes_kolling/* y proporcionando el nombre de usuario y la contraseña cuando se le soliciten.

***Importante**

El código de acceso solo se puede utilizar una vez. Esta suscripción es válida para seis meses después de la activación y no es transferible. Si el código de acceso ya hubiera sido descubierto, es posible que ya no sea válido. En este caso, puede adquirir una suscripción accediendo a *http://www.pearsonhighered.com/barnes_kolling/* y siguiendo las instrucciones que aparecen en pantalla.

Programación orientada a objetos con Java usando BlueJ

5ª edición

Programación orientada a objetos con Java usando BlueJ

5ª edición

David J. Barnes
Michael Kölling
Universidad de Kent

Traducción
Vuelapluma

Revisión técnica

Timothy Read
Departamento de Lenguajes y Sistemas Informáticos. E.T.S.I. Informática
Universidad Nacional de Educación a Distancia (UNED)

Prof. María Ángeles Díaz Fondón, Ph.D.
Departamento de Informática. Escuela de Ingeniería Informática
Universidad de Oviedo

Prof. María Cándida Luengo Díez, Ph.D.
Departamento Informática. Escuela de Ingeniería Informática
Universidad de Oviedo

PEARSON

Datos de catalogación bibliográfica

Programación orientada a objetos con Java usando BlueJ
David J. Barnes, Michael Kölling

PEARSON EDUCACIÓN, S. A. 2013
ISBN: 978-84-8322-791-6
Materia: Informática, 004

Formato: 195 x 250 mm Páginas: 608

Ribera del Loira, 28
28042 Madrid (España)
www.pearson.es

Authorized translation from the English language edition, entitled OBJECTS FIRST WITH JAVA: A PRACTICAL INTRODUCTION USING BLUEJ, 5th Edition by DAVID BARNES; MICHAEL KOLLING, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2012.

ISBN: 978-84-8322-791-6
Depósito Legal: M-87-2013

Equipo de edición
Editor: Miguel Martín-Romo
Técnico editorial: María Vázquez

Equipo de diseño
Diseñadora Senior: Elena Jaramillo
Técnico de diseño: Pablo Hoces de la Guardia

Equipo de producción
Directora: Marta Illescas
Coordinadora: Tini Cardoso

Diseño de cubierta: César de la Morena

Composición: Vuelapluma

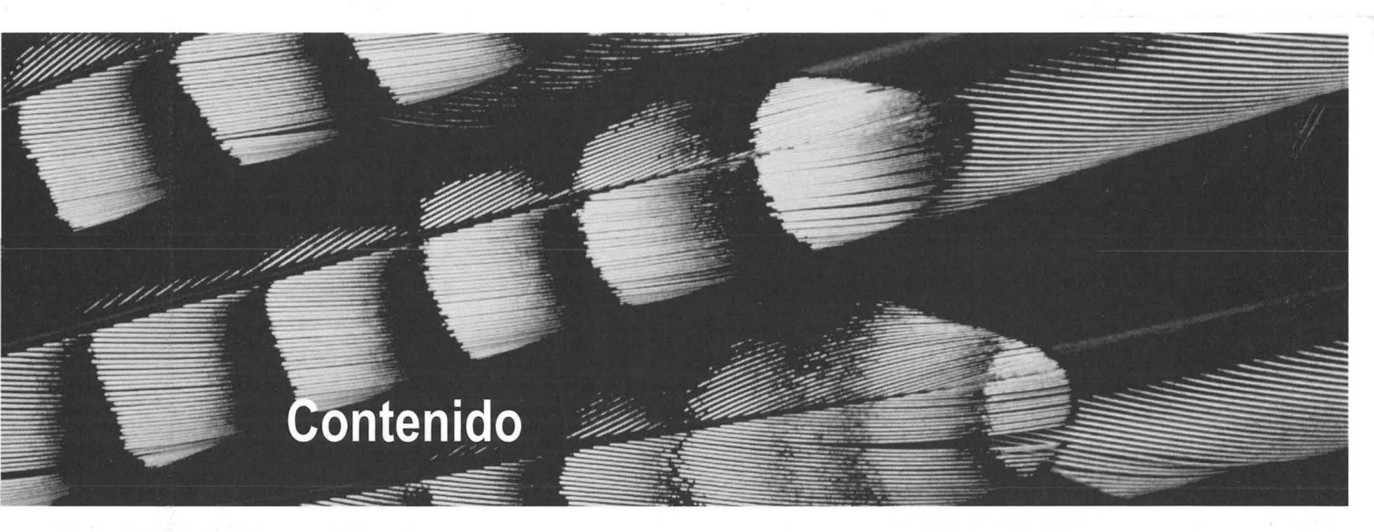
Impreso por:

IMPRESO EN ESPAÑA-*PRINTED IN SPAIN*

Este libro ha sido impreso con papel y tintas ecológicos

A mi esposa Helen,
treinta años y subiendo
djb

A Monica, por todo
mk



Contenido

Prólogo
por James Gosling, creador de Java

Ver cómo mi hija Kate y sus compañeros se peleaban con un curso de Java que utilizaba un IDE comercial fue una experiencia traumática. La sofisticación de la herramienta añadía una gran complejidad a la tarea de aprendizaje. Me habría gustado comprender antes lo que estaba pasando, pero de hecho no fui capaz de hablar con el profesor acerca del problema hasta que ya era demasiado tarde. Este es exactamente el tipo de situación en el que BlueJ proporciona una perfecta solución.

BlueJ es un entorno de desarrollo interactivo con un objetivo muy claro: está diseñado para ser utilizado por estudiantes que estén aprendiendo a programar. Fue diseñado por profesores que se han estado enfrentado a este problema en sus clases de manera cotidiana. Ha sido enormemente instructivo poder hablar con las personas que desarrollaron BlueJ: tienen una idea muy clara de cuál es su objetivo. Las discusiones tendían a centrarse más hacia qué cosas dejar fuera que en qué cosas incorporar. BlueJ es un entorno muy limpio y muy bien enfocado.

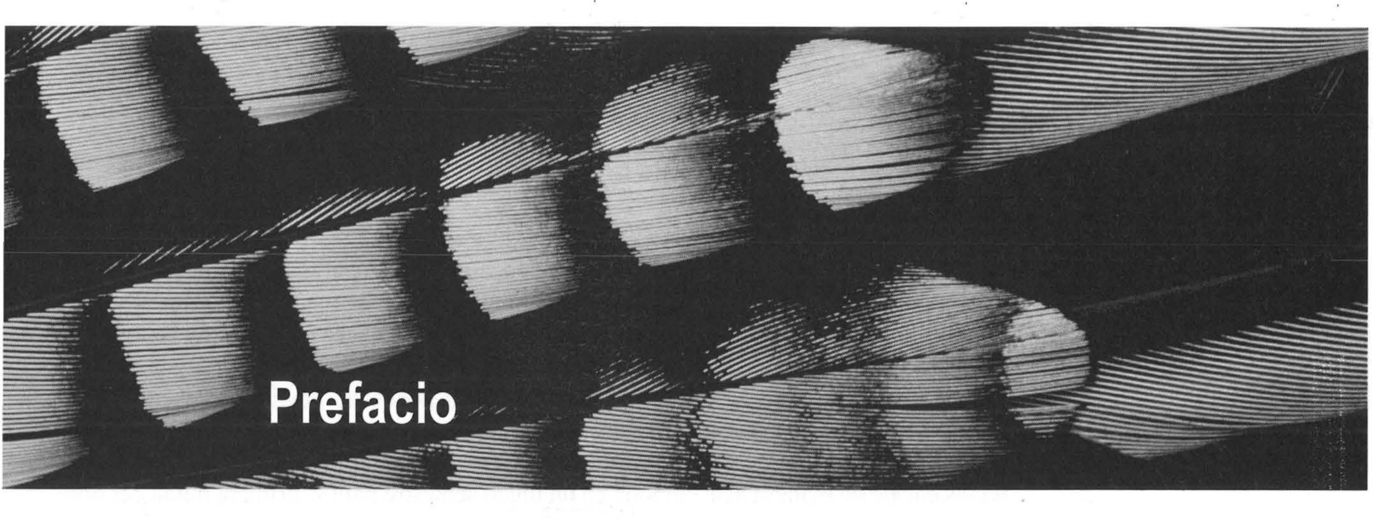
Sin embargo, este libro no se ocupa de BlueJ, sino que es un libro de programación en Java.

A lo largo de los últimos años, Java ha llegado a utilizarse ampliamente en la enseñanza de la programación. Esto se debe a varias razones. Una de ellas es que Java tiene muchas características que hacen que su enseñanza sea muy fácil: tiene una definición relativamente limpia; además, el exhaustivo análisis sintáctico realizado por el compilador informa a los estudiantes muy pronto de los problemas existentes y tiene un modelo de memoria muy robusto que elimina la mayoría de los errores "misteriosos" que surgen cuando se ven comprometidas las fronteras de los objetos o el sistema de tipos. Otra razón es que Java ha llegado a ser comercialmente muy importante.

Este libro aborda desde el principio el concepto más difícil de enseñar: los objetos. Guía a los estudiantes desde los primeros pasos hasta la exposición de algunos conceptos muy sofisticados.

Consigue resolver una de las cuestiones más peliagudas a la hora de escribir un libro sobre programación: cómo manejar la mecánica de escribir y ejecutar en la práctica un programa. La mayor parte de los libros suelen obviar el problema o tratarlo ligeramente, dejando que sea el profesor el que se las apañe para resolver la cuestión y dejándole también el engorro de poner en relación el material que está siendo enseñado con los pasos que los estudiantes deben dar para trabajar en los ejercicios. En lugar de ello, este libro presupone el uso de BlueJ y es capaz de integrar la tarea de comprender los conceptos con la mecánica de cómo deben hacer los estudiantes para explorarlos.

Me hubiera gustado que este libro hubiera estado disponible el año pasado para que mi hija lo hubiera utilizado. Quizá el próximo año tenga mejor fortuna...



Novedades en la quinta edición

Esta es la quinta edición de este libro y hemos aprovechado la oportunidad para incorporar varios cambios importantes respecto a las ediciones anteriores.

- Se han incorporado en todos los lugares apropiados una serie de características de Java 7:
  - Se cubre la "notación de diamante" (inferencia de tipos genéricos) al introducir los genéricos.
  - Se ha añadido el tratamiento de algunas nuevas clases seleccionadas del paquete `nio` para la E/S.
  - Se muestra el uso de las cadenas de caracteres en instrucciones switch.
  - Se aborda la nueva sintaxis de tratamiento de excepciones, incluyendo la multicaptura y el try con recursos.
- Se ha añadido una serie de nuevos e interesantes proyectos utilizando archivos de música y redes sociales. Muchos otros ejemplos han sido modificados, actualizados y mejorados.
- La prueba de unidades ahora está basada en JUnit 4.
- La versión de BlueJ es la 3.0.5 y está disponible en el CD de acompañamiento. Esta versión incluye el coloreado de ámbitos, JUnit 4 y soporte para Java 7.
- Incluye *VideoNotes*, unos cortos tutoriales en vídeo para reforzar los conceptos fundamentales a lo largo del libro.
- Se ha ampliado el tratamiento de las colecciones y de la iteración en el Capítulo 4.
- Se ha añadido el acceso a *Blueroom*, una comunidad y foro para profesores (en inglés) de BlueJ diseñada para la compartición de recursos y la colaboración con los autores y con otros profesores que enseñan utilizando BlueJ.

Algunos de estos cambios son el resultado de la introducción de los cambios del lenguaje en Java 7. Hablamos en esta edición, por ejemplo, de la notación de diamante, del uso de cadenas de caracteres en instrucciones switch, de los cambios en las rutinas de tratamiento de excepciones y de algunas de las clases de `nio`, pero los ejemplos pueden seguir siendo utilizados por aquellos que todavía no se hayan actualizado a Java 7.

La mayoría de los cambios de esta edición son, sin embargo, el resultado de los casi diez años de experiencia que hemos ya acumulado al utilizar este material con nuestros estudiantes, junto con los comentarios recibidos de nuestros colegas profesores y de los lectores. Un ejemplo concreto es la ampliación del tratamiento de las colecciones y la iteración en el Capítulo 4, pero se han hecho otras muchas pequeñas ampliaciones en aquellos lugares donde hemos considerado necesario clarificar temas que requerían algo más de explicación. También hemos cambiado el orden de los Capítulo 6 y 7 para que el flujo de temas encaje de manera más cómoda en un único semestre para la primera mitad del libro.

Hemos introducido varios nuevos proyectos para hacer más atractivo el tratamiento de los temas existentes. Entre ellos, se incluyen un organizador de archivos de música en el Capítulo 4, una tienda en línea en el Capítulo 7 y una red social en los Capítulo 8 y 9.

De todos modos, la concepción y el estilo distintivos de este libro, en los que se basó ya la primera versión, no se han modificado; la razón es que el libro parece, en términos generales, estar "funcionando".

Los comentarios que hemos recibido de los lectores de las ediciones anteriores han sido abrumadoramente positivos, y muchas personas han contribuido a hacer mejor este libro enviando sus comentarios y sugerencias, localizando errores e informándonos de ellos, añadiendo material al sitio web del libro, participando en el foro de discusión o traduciendo el libro a otros idiomas.

Este libro es una introducción para principiantes a la programación orientada a objetos. El libro está enfocado principalmente sobre los conceptos generales de orientación a objetos y de programación, desde una perspectiva de ingeniería del software.

Aunque los primeros capítulos están escritos para estudiantes que no tengan experiencia en programación, los capítulos posteriores son adecuados para estudiantes más avanzados o incluso para programadores profesionales. En particular, los programadores con experiencia en un lenguaje no orientado a objetos que quieran efectuar la migración y adaptar sus habilidades a la orientación a objetos podrán también beneficiarse de la lectura del libro.

A lo largo del libro, utilizamos dos herramientas para poder llevar a la práctica los conceptos presentados: el lenguaje de programación Java y el entorno de desarrollo Java BlueJ.

Java

Elegimos Java por una combinación de dos aspectos: el diseño del lenguaje y su popularidad. El propio lenguaje de programación Java proporciona una implementación muy limpia de la mayor parte de los conceptos de orientación a objetos más importantes, y sirve muy bien como lenguaje introductorio de enseñanza. Además, su popularidad garantiza una inmensa variedad de recursos de soporte.

En cualquier área temática, resulta muy útil tener disponible una diversidad de fuentes de información tanto para los estudiantes como para los profesores. Para Java en concreto, existen infinidad de libros, tutoriales, ejercicios, compiladores, entornos y tests de muchos tipos y en muchos estilos diferentes. Muchos de ellos son recursos en línea y una gran cantidad de los mismos están disponibles de manera gratuita. La gran cantidad y la buena calidad

de los materiales de soporte hace de Java una elección excelente como introducción a la programación orientada a objetos.

Con tanto material Java ya disponible, ¿sigue habiendo necesidad de decir algo más acerca del tema? Nosotros creemos que sí y la segunda herramienta que utilizamos es una de las razones de que pensemos eso...

BlueJ

La segunda herramienta, BlueJ, merece más comentarios. Este libro es original desde el punto de vista de que utiliza de forma completamente integrada el entorno BlueJ.

BlueJ es un entorno de desarrollo Java que está siendo desarrollado y mantenido por el *Computing Education Research Group* de la Universidad de Kent, en Canterbury, Reino Unido, explícitamente como entorno para la enseñanza de la programación orientada a objetos. Está mejor adaptado para la enseñanza de esos conceptos introductorios que otros entornos por varias razones:

- La interfaz de usuario es mucho más simple que otras herramientas. Los estudiantes principiantes pueden normalmente utilizar el entorno BlueJ de una forma competente después de una presentación de 20 minutos. A partir de ahí, la enseñanza puede concentrarse en los conceptos importantes –orientación a objetos y Java–, sin necesidad de perder el tiempo hablando de entornos, sistemas de archivos, rutas de clases o conflictos DLL.
- El entorno soporta importantes herramientas de enseñanza que no están disponibles en otros entornos. Una de ellas es la visualización de la estructura de clases. BlueJ muestra automáticamente un diagrama de tipo UML que representa las clases del proyecto y sus relaciones. Visualizar estos conceptos es de gran ayuda tanto para los profesores como para los estudiantes. ¡Es difícil comprender el concepto de objeto cuando lo único que ves en la pantalla son líneas de código! La notación de diagramas es un subconjunto simple de UML, que está adaptado, de nuevo, a las necesidades de los estudiantes principiantes. Esto hace que sea fácil de entender, permitiendo al mismo tiempo la migración a la notación UML completa en cursos posteriores.
- Una de las principales ventajas del entorno BlueJ es la capacidad del usuario para crear directamente objetos de cualquier clase y luego interactuar con sus métodos. Esto da la oportunidad de experimentar directamente con los objetos sin añadir complicaciones innecesarias al entorno. Los estudiantes pueden casi "sentir" qué quiere decir crear un objeto, llamar a un método, pasar un parámetro o recibir un valor de retorno. Pueden probar un método inmediatamente después de haberlo escrito, sin necesidad de escribir programas de prueba. Esta funcionalidad resulta inestimable a la hora de entender los conceptos subyacentes y los detalles del lenguaje.
- BlueJ incluye muchas otras herramientas y características que están específicamente diseñada para quienes están aprendiendo a desarrollar software. Algunas de ellas pretenden ayudar a entender los conceptos fundamentales (como por ejemplo la funcionalidad de resaltado de ámbitos en el editor), mientras que otras están diseñadas para introducir herramientas y técnicas adicionales, como las pruebas integradas mediante JUnit o el trabajo en equipo mediante un sistema de control de versiones, como Subversion, una vez que los estudiantes estén listos. Varias de estas características son originales del entorno BlueJ.

BlueJ es un entorno de Java completo. No se trata de una versión simplificada y recortada de Java para el entorno académico. Se ejecuta sobre el *Java Development Kit* de Oracle y hace uso del compilador y la máquina virtual estándar. Esto garantiza que siempre se adapte a la especificación oficial y más actualizada de Java.

Los autores de este libro tienen muchos años de experiencia docente con el entorno BlueJ (y muchos más años de experiencia antes de eso). Ambos hemos podido comprobar cómo el uso de BlueJ ha hecho incrementarse el interés, la comprensión y la actividad de los estudiantes en nuestros cursos. Uno de los autores es también desarrollador del sistema BlueJ.

Primero los objetos reales

Una de las razones para elegir BlueJ fue que permite un enfoque en el que los profesores verdaderamente tratan en primer lugar con los conceptos importantes. La frase "primero los objetos" ha sido un caballo de batalla durante algún tiempo para muchos profesores y autores de libros de texto. Lamentablemente, el lenguaje Java no hace que resulte muy sencillo este noble objetivo. Es necesario lidiar con numerosos engorros de detalle y relativos a la sintaxis antes de poder tener la primera experiencia con un objeto real. El programa Java mínimo para poder crear e invocar un objeto típicamente implica:

- Escribir una clase.
- Escribir un método principal, incluyendo conceptos tales como métodos estáticos, parámetros y matrices en la signatura.
- Una instrucción para crear el objeto ("new").
- Una asignación a una variable.
- La declaración de la variable, incluyendo su tipo.
- Una llamada a método, utilizando la notación de punto.
- Posiblemente una lista de parámetros.

Como resultado, los libros de texto suelen hacer una de dos cosas:

- progresar a través de esa aterradora lista de conceptos y solo comenzar a tocar los objetos alrededor más o menos del Capítulo 4, o
- utilizar un programa de estilo "Hello, world" con un único método estático principal como primer ejemplo, sin crear ningún objeto en absoluto.

Con BlueJ, esto no es un problema. ¡El estudiante puede crear un objeto e invocar sus métodos como primera actividad! Puesto que los usuarios pueden crear objetos directamente e interactuar con ellos, pueden exponerse fácilmente conceptos tales como clases, objetos, métodos y parámetros de una manera concreta antes incluso de echar un vistazo a la primera línea de sintaxis Java. En lugar de explicar aquí más cosas acerca de este aspecto, sugerimos al lector interesado que se sumerja en el Capítulo 1 –las cosas quedarán claras allí muy rápidamente.

Un enfoque iterativo

Otro aspecto importante de este libro es que sigue un estilo iterativo. En la comunidad docente de la informática hay un patrón de diseño educativo muy conocido que afirma que los con-

ceptos importantes deben enseñarse muy pronto y muy a menudo.[1] Es muy tentador para los autores de libro de texto tratar de decir todo sobre un tema en el mismo lugar donde ese tema se presenta por primera vez. Por ejemplo, es habitual, al introducir los tipos proporcionar una lista completa de datos predefinidos, o explicar todos los tipos de bucles disponibles al explicar el concepto de bucle.

Estos dos enfoques entran en conflicto: no podemos concentrarnos en explicar primero los conceptos importantes y al mismo tiempo proporcionar una cobertura completa de todos los temas con los que nos encontremos. Nuestra experiencia con los libros de texto nos dice que buena parte de los detalles constituyen inicialmente una distracción y tienen el efecto de diluir los puntos importantes, haciendo así que resulten más difíciles de entender.

En este libro, tocamos todos los temas importantes varias veces, tanto dentro de un mismo capítulo como a lo largo de varios capítulos distintos. Los conceptos suelen introducirse con el nivel de detalle necesario para poderlos comprender y aplicar a la tarea que nos traigamos entre manos. Posteriormente, se vuelven a contemplar en un contexto distinto, y la comprensión del concepto se va profundizando a medida que el lector va leyendo los capítulos. Este enfoque también ayuda a resolver el problema de las dependencias mutuas entre conceptos.

Algunos profesores pueden no estar familiarizados con este enfoque iterativo. Examinando los primeros capítulos, los profesores acostumbrados a realizar una introducción más secuencial se sorprenderán al ver la gran cantidad de conceptos que se presentan de manera muy temprana. A primera vista puede parecer que esto requiere una curva de aprendizaje muy pronunciada.

Pero es importante comprender que las cosas no son así. No se espera que los estudiantes comprendan todos los aspectos de un concepto inmediatamente. En lugar de ello, los conceptos fundamentales serán repasados una y otra vez a lo largo del libro, permitiendo que los estudiantes desarrollen una comprensión cada vez más profunda con el paso del tiempo. Puesto que su nivel de conocimientos varía a medida que van progresando, la revisión posterior de conceptos importantes permite obtener una mejor comprensión global.

Hemos comprobado este enfoque con los estudiantes muchas veces. Y parece que los estudiantes tienen menos problemas a la hora de asumir este enfoque que algunos profesores muy experimentados. Y recuerde: ¡una curva de aprendizaje muy pronunciada no es ningún problema, en tanto seamos capaces de garantizar que nuestros estudiantes puedan ascender por ella!

Tratamiento no exhaustivo del lenguaje

Relacionada con el enfoque iterativo está la decisión de no intentar proporcionar un tratamiento completo del lenguaje Java dentro del libro.

El objetivo principal del libro es transmitir principios de programación orientada a objetos en general, no detalles del lenguaje Java en concreto. Los estudiantes que utilicen este libro pueden estar trabajando como profesionales del software durante los próximos 30 o 40 años de su vida –así que podemos apostar con casi total seguridad que la mayor parte de su trabajo no se realizará en Java. Todo libro de texto serio debe, por supuesto, intentar prepararles para cosas más fundamentales que el lenguaje que esté actualmente de moda.

---

[1] El patrón denominado *Early Bird* en J. Bergin: "Fourteen pedagogical patterns for teaching computer science", *Proceedings of the Fifth European Conference on Pattern Languages of Programs* (EuroPLop 2000), Irsee, Alemania, julio 2000.

Por otro lado, muchos detalles de Java son importantes para poder realizar las tareas prácticas. En este libro, exponemos las estructuras del lenguaje Java con todo el detalle necesario para ilustrar los conceptos que estemos presentando y para implementar los trabajos prácticos. Algunas estructuras específicas de Java se han dejado deliberadamente fuera de las explicaciones.

Somos conscientes de que algunos profesores decidirán tratar algunos temas que nosotros no exponemos en detalle. Eso es algo perfectamente esperable y necesario. Sin embargo, en lugar de tratar de cubrir nosotros mismos todos los temas posibles (haciendo así que el tamaño de este libro fuera de 1500 páginas), lo que hemos preferido hacer es utilizar *enganches*. Esos enganches son punteros, a menudo presentados en forma de preguntas que plantean el tema y proporcionan referencias a un apéndice o a material externo al libro. Estos enganches aseguran que cada tema relevante sea planteado en el momento apropiado, dejando al lector o al profesor la decisión de con qué nivel de detalle cubrir cada tema. Por tanto, los enganches sirven como recordatorio de la existencia de cada tema y como marcador que indica el punto de la secuencia en el que pueden insertarse las correspondientes explicaciones.

Un profesor puede decidir utilizar el libro tal como está, siguiendo la secuencia que sugerimos, o introducir algunas de las digresiones sugeridas por algunos de los enganches incluidos en el texto.

A menudo, los capítulos incluyen también varias cuestiones en las que se sugiere material formativo relacionado con el tema, pero que no se trata en el libro. Confiamos en que los profesores expliquen algunas de estas cuestiones en el aula o que los estudiantes investiguen las respuestas como parte de las tareas que deben resolver en casa.

Enfoque basado en proyectos

La introducción del material se realiza basándose en proyectos. Se exponen numerosos proyectos de programación y se proporcionan muchos ejercicios. En lugar de introducir una nueva estructura y luego proporcionar un ejercicio para aplicar esa estructura con el fin de resolver una tarea, primero planteamos un objetivo y un problema. El análisis del problema permite determinar el tipo de soluciones que necesitamos. Como consecuencia, se introducen las estructuras del lenguaje a medida que son necesarias para resolver los problemas que tenemos encima de la mesa.

Los primeros capítulos proporcionan al menos dos ejemplos para el análisis. Se trata de proyectos que se analizan en detalle con el fin de ilustrar los conceptos importantes de cada capítulo. La utilización de dos ejemplos muy distintos apoya nuestro uso del enfoque iterativo: cada concepto vuelve a analizarse en un contexto distinto, después de haberlo presentado.

Al diseñar este libro hemos tratado de utilizar un gran número y una amplia variedad de diferentes ejemplos de proyectos. Esperamos que esto sirva para captar el interés del lector, pero también ayuda a ilustrar la variedad de contextos distintos en los que pueden aplicarse los conceptos. Resulta complicado encontrar buenos proyectos de ejemplo y esperamos que los que nosotros hemos seleccionado sirvan para proporcionar a los profesores un buen punto de partida y muchas ideas para una amplia variedad de definición de tareas para los estudiantes.

La implementación de todos nuestros proyectos se ha escrito de manera muy cuidadosa, de forma que puedan estudiarse muchas cuestiones periféricas leyendo el código fuente de los

proyectos. Creemos firmemente en la ventaja de aprender leyendo e imitando buenos ejemplos. Sin embargo, para que esto funcione, hay que asegurarse de que los ejemplos que lean los estudiantes estén bien escritos y merezcan la pena de ser imitados. Hemos hecho todos los esfuerzos para que esto sea así.

Todos los proyectos están diseñados como problemas no cerrados. En el libro se analizan en detalle una o más versiones de cada problema, los proyectos están diseñados para que los estudiantes puedan acometer proyectos de ampliación y mejora. El libro incluye el código fuente completo de todos los proyectos. En la página xxv se proporciona una lista de todos los proyectos utilizados en el libro.

Secuencia de conceptos en lugar de estructuras del lenguaje

Otro aspecto que distingue a este libro de muchos otros es que está estructurado según una serie de tareas fundamentales de desarrollo de software y no necesariamente de acuerdo con las estructuras concretas del lenguaje Java. Un indicador de esto son los títulos de los capítulos. En este libro no encontrará muchos de los títulos de capítulo tradicionales, como "Tipos de datos primitivos" o "Estructuras de control". La estructuración mediante tareas fundamentales de desarrollo nos permite proporcionar una introducción mucho más general que no está limitada por los detalles del lenguaje de programación concreto utilizado. También pensamos que a los estudiantes les resulta más fácil comprender la motivación de este curso introductorio, y que también hace que la lectura resulte mucho más interesante.

Como resultado de este enfoque, es más difícil emplear este libro como texto de referencia. Los libros de texto introductorios y los libros de referencia tienen objetivos distintos y en parte contrapuestos. Hasta un cierto grado, un libro puede tratar de ser ambas cosas, pero en algún momento hay que adoptar ciertos compromisos. Nuestro libro está diseñado, claramente, como libro de texto, y cada vez que nos hemos encontrado con un conflicto, el estilo propio ha tenido precedencia sobre su uso como libro de referencia.

Sin embargo, sí que hemos proporcionado soporte para su uso como libro de referencia enumerando al principio de cada capítulo la lista de estructuras Java que en él se presentan.

Secuencia de capítulos

El Capítulo 1 trata con los conceptos más fundamentales de la orientación a objetos: objetos, clases y métodos. Proporciona una introducción sólida y práctica a estos conceptos sin entrar en los detalles de la sintaxis Java. Presentamos brevemente el concepto de abstracción por primera vez. Este será, necesariamente, uno de los hilos conductores que abarca varios capítulos. El Capítulo 1 también echa un primer vistazo a algo de código fuente. Lo hacemos utilizando un ejemplo de formas gráficas que se pueden dibujar de manera interactiva y un segundo ejemplo de un sistema sencillo de matriculación en clases de laboratorio.

El Capítulo 2 abre el tema de las definiciones de clases e investiga cómo se escribe el código fuente Java para definir el comportamiento de los objetos. Explicamos cómo definir campos e implementar métodos y señalamos el papel crucial que el constructor tiene a la hora de configurar el estado de un objeto, definido por sus campos. Aquí introducimos también los primeros tipos de instrucciones. El ejemplo principal es una implementación de una máquina expendedora. También se vuelve a examinar el ejemplo de las clases de laboratorio del Capítulo 1 para profundizar un poco más en él.

El Capítulo 3 abre entonces el panorama para explicar la interacción entre varios objetos. En él podemos ver cómo colaboran los objetos invocando los métodos de otros, con el fin de realizar una tarea común. También se explica cómo puede un objeto crear otros objetos. Se analiza un ejemplo consistente en una pantalla de un reloj digital que utiliza dos objetos de pantalla numérica para mostrar las horas y los minutos. Una versión del proyecto en la que se incluye una interfaz GUI ejemplifica uno de los temas recurrentes del libro –para el que a menudo proporcionamos código adicional con el fin de que el estudiante más interesado y capacitado pueda explorarlo, sin analizarlo detalladamente en el texto. Como segundo ejemplo principal, examinamos una simulación de un sistema de correo electrónico, en el que los clientes de correo pueden intercambiarse mensajes.

En el Capítulo 4 continuamos construyendo estructuras más amplias de objetos y volvemos a tocar los temas de la abstracción y de la interacción de objetos introducidos en los capítulos anteriores. Lo más importante es que se comienzan a utilizar colecciones de objetos. En el capítulo se implementa un organizador de archivos de música y un sistema de subastas, con el fin de presentar las colecciones. Al mismo tiempo se explica la iteración a través de las colecciones y se echa un primer vistazo a los bucles *for-each* y *while*. La primera colección que se utiliza es un `ArrayList`. En la segunda mitad del capítulo, presentamos las matrices como forma especial de colección y el bucle *for* como otra forma de bucle. Como ejemplo de utilización de las matrices se estudia la implementación de un analizador de un registro web.

El Capítulo 5 se ocupa de las librerías e interfaces. Se presenta la librería Java y se exponen algunas clases de librería importantes. Lo fundamental es que se explica cómo leer y comprender la documentación de las librerías. Se explica también la importancia de escribir documentación en los proyectos de desarrollo software y se termina practicando el modo de escribir una documentación adecuada para nuestras propias clases. `Random`, `Set` y `Map` son ejemplos de clases con los que nos encontraremos en este capítulo. Implementamos un sistema de diálogo de tipo *Eliza* y una simulación gráfica de una pelota que rebota en pantalla con el fin de aplicar esas clases.

En el Capítulo 6 analizamos de manera más formal la cuestión de dividir un dominio de problema en una serie de clases para la implementación. Presentamos el problema de diseñar correctamente las clases, incluyendo conceptos tales como el diseño dirigido por responsabilidad, el acoplamiento, la cohesión y la refactorización. Para estas explicaciones se emplea un juego de aventuras interactivo basado en texto (*World of Zuul*). A lo largo del capítulo se efectúan varias iteraciones en las que se va mejorando la estructura interna de las clases del juego y ampliando su funcionalidad, para terminar con una larga lista de propuestas de ampliación que pueden ser asignadas como proyectos a los estudiantes.

El Capítulo 7, titulado "Objetos con un buen comportamiento", trata con un grupo completo de problemas relacionados con la producción de clases correctas, comprensibles y mantenibles. Se abordan cuestiones que van desde la escritura de un código claro y comprensible –incluyendo el estilo y los comentarios– hasta las pruebas y la depuración. Se introducen las estrategias de prueba, incluidas las pruebas formales de regresión mediante JUnit, y se analizan en detalle varios métodos de depuración. Utilizamos un ejemplo de una tienda en línea y una implementación de una calculadora electrónica para explicar estos temas.

Los Capítulos 8 y 9 introducen la herencia y el polimorfismo, junto con muchas de las cuestiones de detalle relacionadas. Para ilustrar los conceptos se analiza una parte de una red social. Se explican en detalle los temas de la herencia de código, los subtipos, las llamadas a métodos polimórficos y la sustitución de métodos.

En el Capítulo 10 se implementa una simulación predador-presa, lo que sirve para analizar mecanismos adicionales de abstracción basados en la herencia, y en particular las interfaces y las clases abstractas.

El Capítulo 11 desarrolla un visualizador de imágenes y una interfaz gráfica de usuario para el organizador de música (Capítulo 4). Ambos ejemplos sirven para explicar cómo construir interfaces gráficas de usuario (GUI).

El Capítulo 12 entra entonces en la difícil cuestión de cómo manejar los errores. Se analizan varios posibles problemas y sus soluciones, y se presenta de manera detallada el mecanismo de tratamiento de excepciones en Java. Para ilustrar los conceptos se amplía y mejora una aplicación de libreta de direcciones. Como caso de estudio en el que el tratamiento de errores es un requisito esencial se utiliza el tema de la entrada/salida.

El Capítulo 13 da un paso atrás para analizar en detalle el siguiente nivel de abstracción: cómo estructurar en clases y métodos un problema descrito de manera vaga. En los capítulos anteriores, hemos asumido que ya existían grandes partes de la estructura de la aplicación, y lo que hacíamos era realizar mejoras de las mismas. Ahora es el momento de explicar como empezar partiendo de cero. Esto implica un análisis detallado de qué clases hay que utilizar para implementar nuestra aplicación, cómo interactúan y cómo deberían distribuirse las responsabilidades. Utilizamos tarjetas CRC (Clases, Responsabilidades, Colaboradores) para abordar este problema, mientras diseñamos un sistema de reserva de entradas de cine.

En el Capítulo 14 tratamos de unir todo e integrar muchos de los temas anteriores del libro. Se trata de un caso completo de estudio, comenzando con el diseño de la aplicación, siguiendo con el de las interfaces de las clases y discutiendo muchas de las características funcionales y no funcionales más importantes y los detalles de implementación. Se vuelven a aplicar en un nuevo contexto algunos temas expuestos en capítulos anteriores (tales como la fiabilidad, las estructuras de datos, el diseño de clases y la ampliabilidad).

Suplementos



VideoNotes. VideoNotes es la nueva herramienta visual de Pearson diseñada para enseñar a los estudiantes técnicas y conceptos de programación clave. Estos cortos vídeos (en inglés) paso a paso muestran cómo resolver problemas que van desde el diseño a la codificación. VideoNotes permite dosificarse uno mismo la formación con unas fáciles herramientas de navegación que incluyen las posibilidades de seleccionar, reproducir, rebobinar, realizar un avance rápido y detenerse dentro de cada ejercicio VideoNote.

VideoNotes está disponible en http://www.pearsonhighered.com/barnes_kolling. Con la compra de cada nuevo libro de texto se incluyen seis meses de acceso prepagado. Si el de código de acceso ya ha sido revelado, puede que ya no sea válido. En ese caso, puede adquirir una suscripción accediendo a **http://www.pearsonhighered.com/barnes_kolling/** y siguiendo las instrucciones mostradas en pantalla.

CD de recursos del estudiante. Este libro incluye un CD con todos los proyectos utilizados como ejemplos de análisis y ejercicios. El CD incluye también el entorno de desarrollo en Java (JDK) y BlueJ para varios sistemas operativos.

Sitio web de acompañamiento para los estudiantes. En el sitio web de acompañamiento, que puede encontrar en **http://www.pearsonhighered.com/barnes_kolling** hay disponibles los siguientes recursos para todos los lectores del libro:

- Guía de estilo de programación para todos los ejemplos del libro.
- Vínculos a material de interés adicional.
- Código fuente completo de todos los proyectos.

Grupo de discusión para los estudiantes. Los estudiantes que quieran plantear preguntas o discutir temas relacionados con el material cubierto en este libro o, con BlueJ en general, pueden hacerlo en http://groups.google.com/group/bluej-discuss en el grupo *bluej-discuss*.

Recursos del profesor. Los siguientes suplementos están disponibles solo para los profesores cualificados:

- Soluciones a los ejercicios del final de cada capítulo.
- Diapositivas PowerPoint.

Visite el *Pearson Instructor Resource Center* en www.pearsonhighered.com/irc para registrarse de cara a poder acceder o contacte con su representante local de Pearson.

Sitio web de los autores. Además del sitio web de acompañamiento del libro creado por el editor, mantenemos un sitio web en http://www.bluej.org/objects-first. En este sitio web podrá encontrar actualizaciones de los ejemplos del libro junto con material adicional. Por ejemplo, la guía de estilo utilizada para todos los ejemplos del libro está disponible en formato electrónico en el sitio web, para que los profesores puedan modificarla con el fin de satisfacer sus propios requisitos. Este sitio web no está soportado por el editor.

Blueroom

Quizá más importante que los recursos estáticos del sitio web es un foro comunitario (en inglés) muy activo para los profesores que enseñan con BlueJ y este libro. Se denomina *Blueroom* y está disponible en

```
http://blueroom.bluej.org
```

Blueroom contiene una colección de recursos con muchos recursos formativos compartidos con otros profesores, así como un foro de discusión en el que los profesores pueden plantear preguntas, debatir temas y permanecer actualizados en lo que respecta a los desarrollos más recientes. En Blueroom, podrá contactar con muchos otros profesores, así como con los desarrolladores de BlueJ y los autores de este libro.



Lista de proyectos explicados en detalle en el libro

figures .... **Capítulo 1**

Dibujo sencillo con algunas formas geométricas; ilustra la creación de objetos, la invocación de métodos y los parámetros.

house .... **Capítulo 1**

Un ejemplo que utiliza objetos que representan formas para dibujar una imagen; introduce el código fuente, la sintaxis de Java y la compilación.

lab-classes .... **Capítulos 1, 2 y 8**

Un ejemplo simple con clases de estudiantes; ilustra los objetos, campos y métodos. Se utiliza de nuevo en el Capítulo 8 para añadir la herencia.

ticket-machine .... **Capítulo 2**

Una simulación de una máquina expendedora de billetes de tren; introduce más conceptos acerca de los campos, constructores, métodos selectores y mutadores, parámetros y algunas instrucciones simples.

book-exercise .... **Capítulo 2**

Almacenamiento de detalles de un libro. Refuerza las estructuras utilizadas en el ejemplo de la máquina expendedora.

clock-display .... **Capítulo 3**

Una implementación de una pantalla para un reloj digital; ilustra los conceptos de abstracción, modularización e interacción de objetos. Incluye una versión con una GUI animada.

mail system .... **Capítulo 3**

Una simulación simple de un sistema de correo electrónico. Se utiliza para ilustrar la creación de objetos y la interacción entre objetos.

music-organizer .... **Capítulos 4 y 11**

Una implementación de un organizador de pistas de música; se utiliza para presentar las colecciones y los bucles. Incluye la posibilidad de reproducir archivos MP3. En el Capítulo 11 se añade una interfaz GUI.

auction .... **Capítulo 4**

Un sistema de subastas. Más conceptos sobre colecciones y bucles, esta vez con iteradores.

weblog-analyzer .... **Capítulo 4**

Un programa para analizar archivos de registros de acceso web; introduce las matrices y los bucles *for*.

tech-support .... **Capítulo 5**

Una implementación de un programa de diálogo de tipo *Eliza* utilizado para proporcionar "soporte técnico" a los clientes; introduce el uso de las clases de librería en general y de algunas específicas en particular, así como la lectura y escritura de documentación.

scribble .... **Capítulo 5**

Un programa de dibujo de formas para dar soporte al aprendizaje acerca de las clases a partir de sus interfaces.

bouncing-balls .... **Capítulo 5**

Una animación gráfica de una serie de bolas rebotando; ilustra la separación interfaz/implementación, así como algunos conceptos gráficos simples.

world-of-zuul .... **Capítulos 6 y 9**

Un juego de aventuras interactivo y basado en texto. Altamente ampliable, constituye una excelente fuente de asignación de proyectos a los estudiantes. Se utiliza aquí para analizar lo que es un buen diseño de clases, así como los conceptos de acoplamiento y cohesión. Se utiliza de nuevo en el Capítulo 9 como ejemplo de uso de la herencia.

online-shop .... **Capítulo 7**

Las primeras etapas de una implementación de una parte de un sitio web de compras en línea, encargada de gestionar los comentarios de los usuarios; se utiliza para analizar las estrategias de prueba y depuración.

calculator .... **Capítulo 7**

Una implementación de una calculadora. Este ejemplo refuerza los conceptos presentados anteriormente y se utiliza para explicar las pruebas y la depuración.

bricks .... **Capítulo 7**

Un ejercicio de depuración simple; modelos en los que se llenan palés con ladrillos para efectuar cálculos simples.

network .... **Capítulos 8 y 9**

Parte de una aplicación de una red social. Se analiza este proyecto y luego se amplía en gran detalle para introducir las bases de la herencia y el polimorfismo.

foxes-and-rabbits .... **Capítulo 10**

Una simulación clásica predador-presa; refuerza los conceptos sobre herencia y añade los de clases abstractas e interfaces.

image-viewer .... **Capítulo 11**

Una aplicación simple de visualización y manipulación de imágenes. Nos concentramos principalmente en la construcción de la GUI.

music-player .... **Capítulo 11**

Se añade una GUI al proyecto *music-organizer* del Capítulo 4, como otro ejemplo de construcción de interfaces GUI.

address-book .... **Capítulo 12**

Una implementación de una libreta de direcciones con una interfaz GUI opcional. La búsqueda es flexible: se pueden buscar las entradas definiendo parcialmente el nombre o el número de teléfono. Este proyecto hace un amplio uso de las excepciones.

cinema-booking-system .... **Capítulo 13**

Un sistema avanzado de gestión de reservas de asientos en un cine. Este ejemplo se utiliza para explicar el descubrimiento de clases y el diseño de aplicaciones. No se proporciona ningún código, ya que el ejemplo representa el desarrollo de una aplicación partiendo de cero.

taxi-company .... **Capítulo 14**

El ejemplo de la compañía de taxis es una combinación de un sistema de reservas, un sistema de gestión y una simulación. Se utiliza como caso de estudio para combinar muchos de los conceptos y técnicas explicados a lo largo del libro.



Agradecimientos

Muchas personas han contribuido de muchas formas distintas a la elaboración de este libro y han hecho posible su creación.

En primer lugar, y como principal agradecimiento tenemos que mencionar a John Rosenberg. John es ahora Vicerrector delegado en la universidad La Trobe en Australia. Es solo pura coincidencia que John no sea uno de los autores de este libro. Él fue una de las fuerzas motrices en el desarrollo de BlueJ y de las ideas y conceptos pedagógicos que subyacen a esta herramienta, desde el principio, y además habíamos estado hablando acerca de escribir este libro durante años. Buena parte del material del libro fue desarrollado en aquellas conversaciones con John. El simple hecho de que los días solo tienen veinticuatro horas, buena parte de las cuales ya están ocupadas por tantas otras tareas, le impidió escribir en la práctica este libro. John ha contribuido de forma significativa a la versión original de este texto y ha ayudado a mejorarlo de muchas formas. Apreciamos inmensamente su amistad y su colaboración.

Muchas otras personas han ayudado a que BlueJ sea lo que es: Bruce Quig, Davin McCall y Andrew Patterson en Australia, e Ian Utting, Poul Henriksen, Neil Brown y Philip Stevens en Inglaterra. Todos ellos han trabajado en BlueJ durante muchos años, mejorando y ampliando el diseño y la implementación, como añadido a sus otras tareas. Sin su trabajo, BlueJ nunca hubiera alcanzado la calidad y la popularidad que tiene hoy día, y este libro no habría podido llegar nunca a ser escrito.

Otra contribución importante que hizo posible la creación de BlueJ y de este libro fue el generoso soporte primero de Sun Microsystems y ahora de Oracle. Sun ha dado soporte a BlueJ durante muchos años y cuando Oracle adquirió Sun este soporte continuó. Estamos enormemente agradecidos por esta contribución crucial.

También nos gustaría dar las gracias a los revisores de esta edición: Daniel Rocco, University of West Georgia; Jeanette Allen, University of West Georgia; Katherine Herbert, Montclair State University; Craig A. Piercy. University of Georgia y Xuemin Chen, Texas Southern University.

El equipo de Pearson también ha hecho un gran trabajo para que este libro viera la luz, lanzarlo al mercado y disipar los peores temores de todo autor –que su libro pueda pasar desapercibido. Queremos dar las gracias en particular a nuestro editor, Tracy Dunkelberger, y a las asistentes editoriales Chelsea Bell y Stephanie Sellinger, que nos han ayudado durante todo el proceso de escritura y producción.

David querría añadir su gratitud personal hacia los profesores y estudiantes del Departamento de Ciencias de la Computación de la Universidad de Kent. Siempre ha sido un privilegio

enseñar a los estudiantes matriculados en el curso de introducción a orientación a objetos. Esos estudiantes proporcionan también el estímulo y la motivación esenciales que hacen que dar clase sea tan divertido. Sin la valiosa ayuda de supervisores de postgrado muy capacitados y motivados, dar las clases serían imposible. Fuera de la vida universitaria, son varias las personas que han proporcionado un entorno social y recreativo que ha impedido que la escritura me absorbiera completamente. En particular, me gustaría dar las gracias a Tim Hopkins y Maggie Bowman –como inagotables fuentes de buena compañía y una diversión ilimitada– y a mis compañeros de The River Band: Ian Lithgow, Mick Budd, Olly Jeffery y Pete Langridge.

Finalmente, me gustaría dar las gracias a mi esposa Helen, cuyo amor es tan especial, y a mis hijos, cuyas vidas son tan maravillosas.

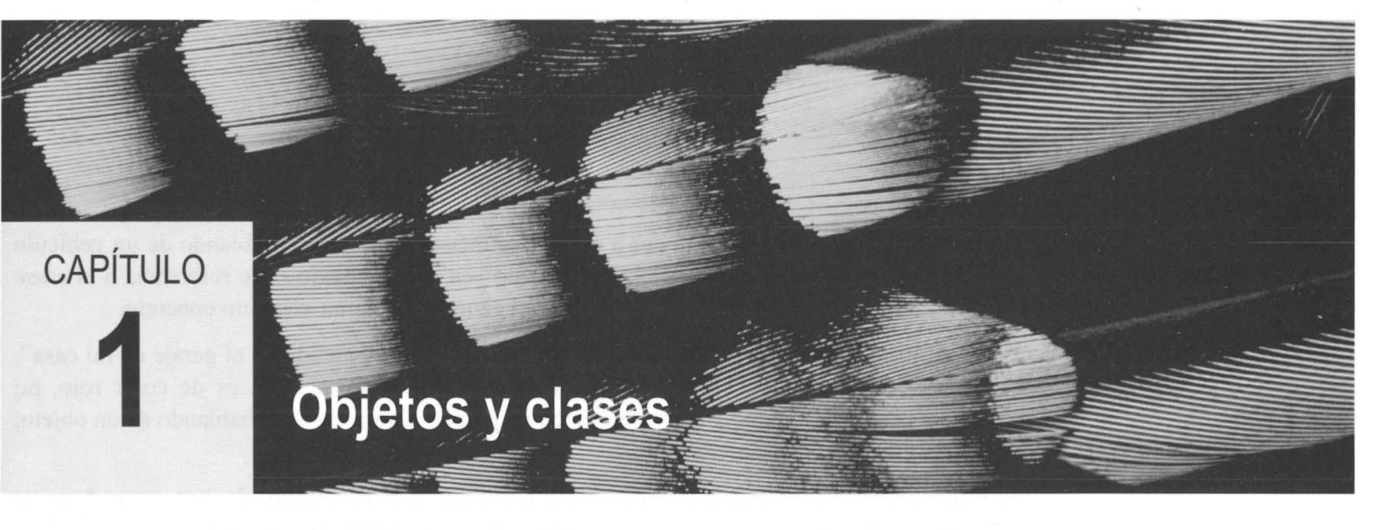
Michael querría dar las gracias a Davin, Neil y Phil, que han hecho un trabajo tan excelente a la hora de construir y mantener BlueJ, Greenfoot y nuestros sitios web comunitarios. Sin ese extraordinario equipo nada de esto podría funcionar.

Tengo que mencionar a mis dos hijitas, Sophie y Feena, que –evidentemente– no están (¿todavía?) terriblemente interesadas en este libro, pero que me hacen continuar esforzándome siempre. Y finalmente, lo más importante, ahí está Monica, el amor de mi vida, no sé dónde estaría sin ella.



Parte 1

Fundamentos de la orientación a objetos



Principales conceptos explicados en el capítulo:

- objetos
- clases
- métodos
- parámetros

Es el momento de acometer la tarea y comenzar con las explicaciones acerca de la programación orientada a objetos. Aprender a programar requiere mezclar algo de teoría con un montón de práctica. En este libro, presentaremos ambas cosas, de manera que se refuercen mutuamente.

En el núcleo del paradigma de la orientación a objetos se encuentran dos conceptos que es necesario comprender antes de poder proseguir con nuestro estudio: *objetos* y *clases*. Estos conceptos forman la base de las tareas de programación en los lenguajes orientados a objetos. Por tanto, vamos a empezar con una breve explicación de estas dos ideas fundamentales.

1.1 Objetos y clases

Concepto:

Los **objetos** Java modelan los objetos pertenecientes a un dominio de problema.

Concepto:

Los objetos se crean a partir de **clases**. La clase describe el tipo de objeto; los objetos representan las instanciaciones individuales de la clase.

Si escribimos un programa informático en un lenguaje orientado a objetos, estaremos creando en nuestro equipo un modelo de una cierta parte del mundo. Los componentes a partir de los cuales se construye el modelo son los *objetos* que aparecen en el dominio del problema concreto que estemos analizando. Esos objetos deben representarse en el modelo informático que estemos desarrollando. Los objetos del dominio del problema varían dependiendo del programa que estemos escribiendo. Puede tratarse de palabras y párrafos si es que estamos programando un procesador de textos, o bien de usuarios y mensajes si estamos trabajando en el sistema de una red social. O puede tratarse de monstruos si estamos escribiendo un juego para computadora.

Los objetos pueden clasificarse y una clase sirve para describir, de una manera abstracta, todos los objetos de un tipo concreto.

Clarifiquemos estas nociones abstractas recurriendo a un ejemplo. Suponga que queremos modelar una simulación de tráfico. En este caso, uno de los tipos de entidad con los que tendremos que tratar son los vehículos. ¿Qué será un vehículo en nuestro contexto: una clase o un objeto? Unas cuantas preguntas nos pueden ayudar a tomar una decisión.

¿De qué color es el vehículo? ¿Qué velocidad puede alcanzar? ¿En qué punto concreto se encuentra en este momento?

Observe que no podemos responder a estas preguntas si no estamos hablando de un vehículo específico. La razón es que la palabra "vehículo" en este contexto hace referencia a la *clase* vehículo; estamos hablando de los vehículos en general, no de un vehículo concreto.

Por el contrario, si hablo de "Mi antiguo vehículo que está aparcado en el garaje de mi casa", sí que podemos responder a las cuestiones planteadas. Ese vehículo es de color rojo, no alcanza una gran velocidad y se encuentra en mi garaje. Ahora estamos hablando de un objeto, de un ejemplo concreto de vehículo.

Normalmente, nos referimos a cada objeto particular con el nombre de *instancia.* A partir de ahora vamos a utilizar con bastante frecuencia este término. La palabra "instancia" es aproximadamente sinónima de "objeto", hablaremos de instancias cuando queramos hacer énfasis en el hecho de que se trata de objetos de una clase concreta (como por ejemplo en la frase "este objeto es una instancia de la clase vehículo").

Antes de continuar con esta discusión más bien teórica, veamos un ejemplo.

1.2 Creación de objetos

Inicie BlueJ y abra el ejemplo denominado *figures.*[1] Verá una ventana similar a la que se muestra en la Figura 1.1.

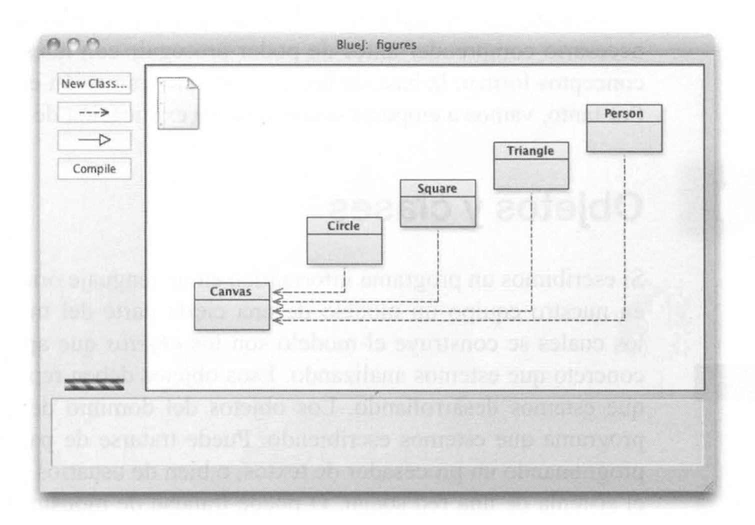


Figura 1.1
El proyecto *figures* en BlueJ.

En esta ventana debería poder ver un diagrama. Cada uno de los rectángulos del diagrama representa una clase de nuestro proyecto. En este proyecto, disponemos de las clases denominadas `Circle`, `Square`, `Triangle` y `Canvas`, que representan círculos, cuadrados, triángulos y lienzos.

Haga clic con el botón derecho del ratón en la clase `Circle` y en el menú emergente seleccione

[1] El lector deberá en bastantes ocasiones realizar actividades y ejercicios mientras lee este libro. Llegados a este punto, vamos a asumir que el lector ya sabe como iniciar BlueJ y abrir los proyectos de ejemplo. Si no es así, lea primero el Apéndice A.

```
new Circle()
```

El sistema pedirá que proporcionemos un "nombre de instancia"; haga clic en OK, ya que el nombre predeterminado que el sistema proporciona es perfectamente válido por el momento. Podrá ver un rectángulo rojo en la parte inferior de la pantalla, etiquetado como "circle1" (Figura 1.2).

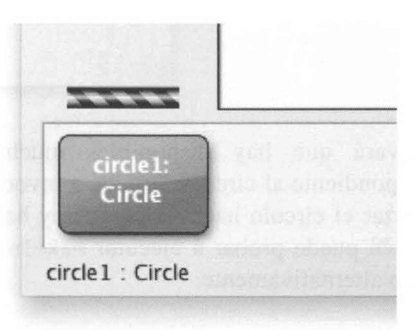


Figura 1.2
Un objeto en el banco de objetos.

Convenio Escribiremos los nombres de las clases con la primera letra en mayúscula (como por ejemplo Circle) y los nombres de los objetos con la primera letra en minúscula (como en circle1). Esto nos ayudará a distinguir de qué estamos hablando.

Ejercicio 1.1 Cree otro círculo. A continuación, cree un cuadrado.

¡Acabamos de crear nuestro primer objeto! "Circle", el icono rectangular de la Figura 1.1, representa la clase `Circle`; `circle1` es un objeto creado a partir de esta clase. El área de la parte inferior de la pantalla en la que se muestra el objeto se denomina *banco de objetos (object bench).*

1.3 Invocación de métodos

Haga clic con el botón derecho del ratón en uno de los objetos círculo (¡no en la clase!) para mostrar un menú emergente con varias operaciones (Figura 1.3). Seleccione `makeVisible` en el menú para hacer visible el objeto; esto hará que aparezca una representación de este círculo en una nueva ventana (Figura 1.4).

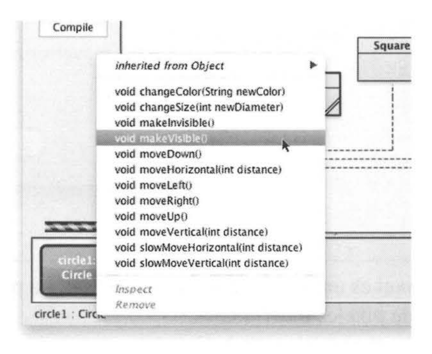


Figura 1.3
Menú emergente de un objeto en el que se indican las operaciones disponibles para el mismo.

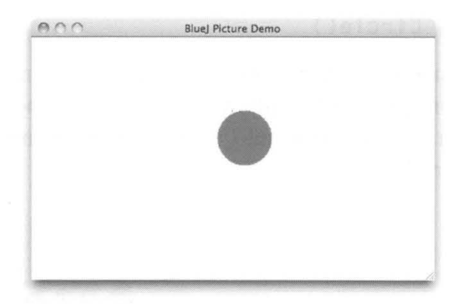


Figura 1.4
Representación de un círculo.

Concepto:

Podemos comunicarnos con los objetos invocando **métodos** sobre los mismos. Si invocamos un método, los objetos normalmente llevan a cabo una acción.

Observará que hay disponibles muchas otras operaciones en el menú emergente correspondiente al círculo. Pruebe a invocar `moveRight` y `moveDown` unas cuantas veces para desplazar el círculo hacia la derecha y hacia abajo, aproximándose al centro de la pantalla. También puede probar a ejecutar `makeInvisible` y `makeVisible` para ocultar y mostrar el círculo alternativamente.

Ejercicio 1.2 ¿Qué sucede si invocamos `moveDown` dos veces? ¿Y si lo invocamos tres veces? ¿Qué sucede si invocamos dos veces `makeInvisible`?

Las entradas que componen el menú emergente del círculo representan operaciones que podemos emplear para manipular el círculo. En Java, estas operaciones se denominan *métodos*. Utilizando la terminología común, decimos que estos métodos se *llaman* o *invocan*. A partir de ahora, emplearemos esta terminología. En consecuencia, a lo largo del texto pediremos que se hagan cosas como "invocar el método `moveRight` de `circle1`."

1.4 Parámetros

Ahora invoque el método `moveHorizontal`. Aparecerá un cuadro de diálogo que le pedirá que introduzca un cierto dato de entrada (Figura 1.5). Escriba 50 y haga clic en OK. Comprobará que el círculo se desplaza 50 píxeles hacia la derecha.[2]

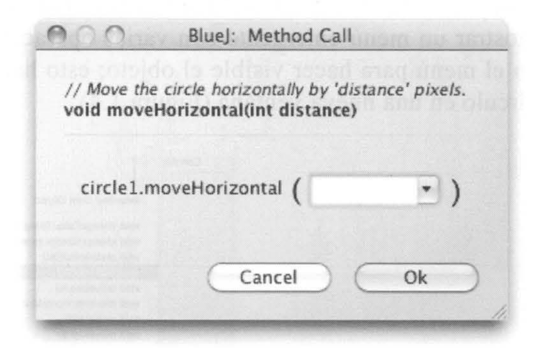


Figura 1.5
Cuadro de diálogo de invocación de un método.

[2] Un pixel es un único punto de la pantalla. La pantalla de la computadora está compuesta por una cuadrícula de píxeles individuales.

Concepto:

Los métodos pueden tener **parámetros** para proporcionar información adicional para una tarea.

El método `moveHorizontal` que acabamos de llamar está escrito de tal forma que requiere algo de información adicional para ejecutarse. En este caso, la información requerida es la distancia, es decir, la especificación de cuánto debe moverse el círculo. Por tanto, el método `moveHorizontal` es más flexible que los métodos `moveRight` y `moveLeft`. Estos dos últimos métodos siempre desplazan el círculo una distancia fija, mientras que `moveHorizontal` nos permite indicar a qué distancia queremos que el círculo se mueva.

Ejercicio 1.3 Invoque los métodos `moveVertical`, `slowMoveVertical` y `changeSize` antes de continuar leyendo. Los dos primeros sirven para desplazar el círculo en vertical, mientras que el último permite cambiar su tamaño. Averigüe también cómo se puede emplear `moveHorizontal` para desplazar el círculo 70 píxeles hacia la izquierda.

Concepto:

A la cabecera de un método se la denomina **signatura**. Proporciona la información necesaria para invocar dicho método.

Los valores adicionales requeridos por algunos métodos se denominan *parámetros*. Cada método indica los tipos de parámetros que necesita. Por ejemplo, cuando se invoca el método `moveHorizontal`, como se muestra en la Figura 1.5, el cuadro de diálogo muestra la correspondiente línea en su parte superior.

```
void moveHorizontal(int distance)
```

Esta líneas se denomina *signatura* del método. La signatura proporciona cierta información acerca del método en cuestión. La parte encerrada entre paréntesis (`int distance`) es la información acerca del parámetro requerido. Para cada parámetro, se definen un *tipo* y un *nombre*. La signatura que acabamos de mostrar indica que el método requiere un parámetro de tipo `int` denominado `distance`. El nombre proporciona una pista acerca del significado de los datos que se espera que se introduzcan.

1.5 Tipos de datos

Concepto:

Los parámetros tienen **tipos**. El tipo define qué clase de valores puede tomar un parámetro.

Un tipo especifica cuáles son los datos que podemos pasar a un parámetro. El tipo `int` hace referencia a los números enteros ("int" es la abreviatura de la palabra inglesa "integer", que significa entero).

En el ejemplo anterior, la signatura del método `moveHorizontal` nos dice que, para poder ejecutar el método, debemos suministrarle un número entero que especifique la distancia que queremos que se desplace el círculo. El campo de entrada de datos mostrado en la Figura 1.5 nos permite introducir dicho número.

En los ejemplos vistos hasta ahora, el único tipo de datos con el que nos hemos encontrado es `int`. Los parámetros de los métodos de desplazamiento y del método `changeSize` son todos ellos de dicho tipo.

Una inspección más detallada del menú emergente de un objeto nos muestra que las entradas del menú correspondientes a los métodos incluyen los tipos de parámetro. Si un método no tiene ningún parámetro, el nombre del método irá seguido por un par de paréntesis vacíos. Si dispone de un parámetro, mostrará el tipo y el nombre de dicho parámetro. En la lista de métodos correspondiente al círculo, solo hay un método con un tipo de parámetro diferente: el método `changeColor` que sirve para cambiar el color del círculo y tiene un parámetro de tipo `String`.

El tipo `String` indica que hace falta una cadena, es decir, una sección de texto (por ejemplo, una palabra o una frase). Las cadenas se encierran siempre entre dobles comillas. Por ejemplo, para introducir la palabra *red* como una cadena y obtener así un círculo de color rojo, escribiríamos:

```
"red"
```

El cuadro de diálogo de invocación del método también incluye una sección de texto denominada *comentario*, por encima de la signatura del método. Los comentarios se incluyen para proporcionar información para el lector (humano) del programa y se describen en el Capítulo 2. El comentario del método `changeColor` describe los nombres de colores que el sistema reconoce.

Ejercicio 1.4 Invoque el método `changeColor` sobre uno de sus objetos círculo e introduzca la cadena `"red"`. Esto debería hacer que cambie el color del círculo. Pruebe con otros colores.

Ejercicio 1.5 Este es un ejemplo muy simple y no son muchos los colores admitidos. Pruebe a ver qué sucede cuando especificamos un color desconocido para el sistema.

Ejercicio 1.6 Invoque el método `changeColor` y escriba el color en el campo de parámetro *sin* las comillas. ¿Qué sucede?

Error común Un error que suelen cometer los principiantes es olvidarse de escribir las dobles comillas al introducir un valor de datos de tipo `String`. Si escribe `green` en lugar de `"green"`, obtendrá un mensaje de error que dirá algo así como "Error: cannot find symbol - variable green." (Error: el sistema no puede encontrar el símbolo - variable green).

Java soporta muchos otros tipos de datos, incluyendo números decimales y caracteres. No vamos a hablar de todos ellos ahora, sino que volveremos sobre esta cuestión más adelante. Si desea saber más cosas ahora acerca de este tema, consulte el Apéndice B.

1.6 Instancias múltiples

Ejercicio 1.7 Cree varios objetos círculo en el banco de objetos. Puede hacerlo seleccionando `new Circle()` en el menú emergente de la clase `Circle`. Hágalos visibles y luego desplácelos por la pantalla utilizando los métodos "move" . Haga que uno de los círculos sea grande y de color amarillo; haga que otro sea pequeño y de color verde. Experimente también con las otras formas geométricas: cree unos cuantos triángulos, cuadrados y personas. Cambie sus posiciones, tamaños y colores.

> **Concepto:**
> **Instancias múltiples.** Pueden crearse varios objetos similares a partir de una única clase.

Una vez que disponemos de una clase, podemos crear tantos objetos (o instancias) de dicha clase como queramos. A partir de la clase `Circle`, podemos crear muchos círculos. A partir de la clase `Square`, podemos crear muchos cuadrados.

Cada uno de esos objetos tiene su propia posición, color y tamaño. Podemos cambiar un atributo de un objeto (como por ejemplo su tamaño) invocando un método sobre dicho objeto. Esto afectará a dicho objeto concreto, pero no a los restantes objetos de la misma clase.

Es posible que el lector también se haya fijado en un detalle adicional acerca de los parámetros. Examine el método `changeSize` del triángulo. Su signatura es

```
void changeSize(int newHeight, int newWidth)
```

Vemos aquí un ejemplo de un método con más de un parámetro. Este método tiene dos parámetros distintos, y se utiliza una coma para separarlos dentro de la signatura. De hecho, los métodos pueden tener cualquier número de parámetros.

1.7 Estado

Concepto:

Los objetos tienen un estado. El **estado** se representa almacenando valores en campos.

El conjunto de valores de todos los atributos que definen a un objeto (como la posición *x*, la posición *y*, el color, el diámetro y el estado de visibilidad de un círculo) se denomina también *estado* del objeto. Este es otro ejemplo de terminología común que emplearemos a partir de ahora.

En BlueJ, el estado de un objeto puede inspeccionarse seleccionando la función *Inspect* en el menú emergente del objeto. Cuando se inspecciona un objeto, se muestra lo que se denomina un *inspector de objetos* (*object inspector*). El inspector de objetos es una vista ampliada del objeto, en la que se muestran los atributos almacenados dentro del mismo (Figura 1.6).

Ejercicio 1.8 Asegúrese de disponer de varios objetos en el banco de objetos y luego inspeccione cada uno de ellos por turnos. Pruebe a cambiar el estado de un objeto (por ejemplo, invocando el método `moveLeft`) mientras que está abierto el inspector de objetos. Podrá ver cómo varían los valores mostrados en el inspector de objetos.

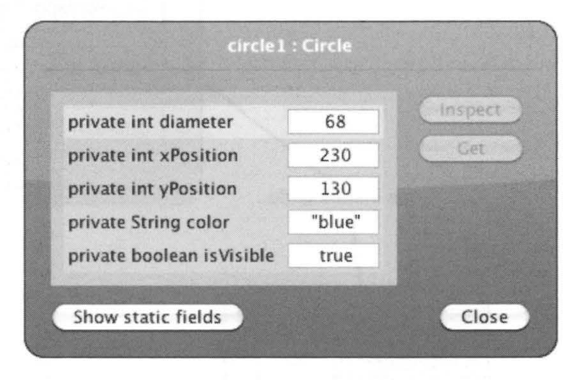


Figura 1.6
Un inspector de objetos, mostrando los detalles de un objeto.

Algunos métodos, al ser invocados, modifican el estado de un objeto. Por ejemplo, `moveLeft` cambia el atributo `xPosition`. Java denomina *campos* a esos atributos de los objetos.

1.8 ¿Qué es lo que contiene un objeto?

Al inspeccionar distintos objetos, observará que todos los objetos de la *misma* clase tienen los mismos campos. Es decir, el número, el tipo y los nombres de los campos son idénticos,

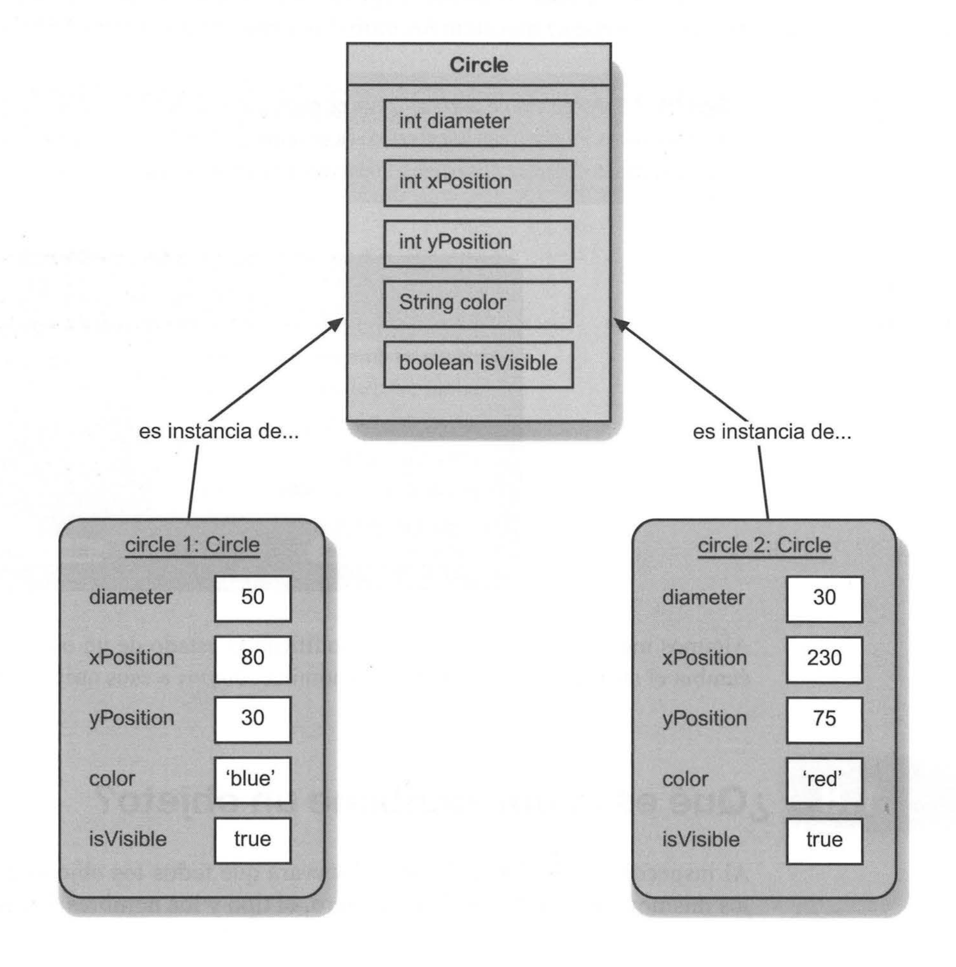
mientras que los valores concretos de cada campo particular de cada objeto pueden ser diferentes. Por el contrario, los objetos de clases *diferentes* pueden tener diferentes campos. Un círculo, por ejemplo, tiene un campo "diameter" (diámetro), mientras que un triángulo tiene campos para la anchura ("width") y la altura ("height").

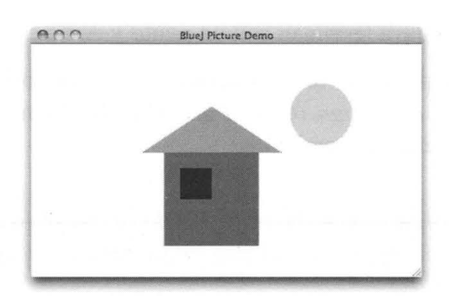
La razón es que el número, los tipos y los nombres de los campos se definen dentro de una clase, no en un objeto. Por tanto, la clase `Circle` define que cada objeto círculo tendrá cinco campos denominados `diameter`, `xPosition`, `yPosition`, `color` e `isVisible`. También define los tipos de esos campos. Es decir, especifica que los tres primeros son de tipo `int`, mientras que `color` es de tipo `String` y el indicador `isVisible` es de tipo `boolean`. (`Boolean` es un tipo que puede representar dos valores: `true` y `false`, que representan los valores lógicos verdadero y falso. Veremos este tema más en detalle más adelante).

Cuando se crea un objeto de la clase `Circle`, el objeto tendrá automáticamente esos campos. Los valores de los campos se almacenarán en el objeto. Esto garantiza que cada círculo tenga un color, por ejemplo, y que los distintos círculos puedan tener colores diferentes (Figura 1.7).

Lo mismo cabe decir de los métodos. Los métodos se definen en la clase del objeto. Como resultado, todos los objetos de una misma clase tendrán los mismos métodos. Sin embargo, los métodos se invocan sobre objetos concretos. Esto hace que esté claro qué objeto es el que hay que modificar cuando, por ejemplo, se invoca un método `moveRight`.

Figura 1.7
Una clase y sus objetos, con sus campos y valores.





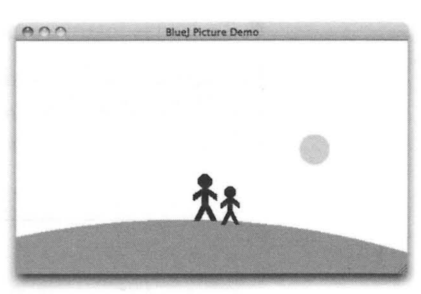


Figura 1.8
Dos imágenes creadas a partir de un conjunto de objetos que representan formas geométricas.

Ejercicio 1.9 La Figura 1.8 muestra dos imágenes diferentes. Seleccione una de ellas y vuelva a crearla utilizando las formas geométricas proporcionadas en el proyecto *figures*. Mientras hace esto, escriba los pasos que ha tenido que dar para conseguirlo. ¿Podría hacerse de diferentes maneras?

1.9 Código Java

Cuando programamos en Java, lo que hacemos, esencialmente, es escribir instrucciones para invocar métodos sobre los objetos, al igual que acabamos de hacer anteriormente con nuestros objetos de figuras geométricas. Sin embargo, al programar en Java no hacemos esto interactivamente, seleccionando métodos en un menú con el ratón, sino que escribimos los comandos correspondientes en forma textual. Podemos ver el aspecto de esos comandos en forma textual utilizando el Terminal de BlueJ.

Ejercicio 1.10 Seleccione *Show Terminal* (Mostrar terminal) en el menú *View* (Ver). Aparecerá otra ventana que BlueJ utiliza para la salida de texto. A continuación, seleccione *Record method calls* (Registrar llamadas a métodos) en el menú *Options* (Opciones) del terminal. Esta función hará que todas nuestras llamadas a métodos (en su forma textual) se escriban en el terminal. Ahora cree unos cuantos objetos, invoque algunos de sus métodos y observe la salida en la ventana del terminal.

Utilizando la función *Record method calls* del terminal, podemos ver que la secuencia de crear un objeto persona e invocar sus métodos `makeVisible` y `moveRight` tiene el siguiente formato textual en Java:

```
Person person1 = new Person();
person1.makeVisible();
person1.moveRight();
```

Podemos observar aquí varias cosas:

- Podemos ver en qué consiste el proceso de creación y de denominación de un objeto. Técnicamente, lo que estamos haciendo es *almacenar el objeto Person en una variable*; hablaremos de esto en detalle en el siguiente capítulo.
- Podemos ver que, para llamar a un método de un objeto, lo que hacemos es escribir el nombre del objeto seguido de un punto y seguido del nombre del método. El comando termina con una lista de parámetros, o con un par de paréntesis vacíos si no hay parámetros.

- Todas las instrucciones Java terminan con un punto y coma.

En lugar de examinar simplemente las instrucciones Java, también podemos escribirlas. Para ello, utilizamos el *bloque de código (Code Pad)*. (Puede detener la función *Record method calls* en este momento y cerrar el terminal.)

Ejercicio 1.11 Seleccione *Show Code Pad* (Mostrar teclado de código) en el menú *View*. Se mostrará un nuevo panel junto al banco de objetos en la ventana principal de BlueJ. Este panel es el *Code Pad*. En él puede escribirse código Java.

En el teclado de código, podemos escribir código Java que haga las mismas cosas que antes hemos llevado a cabo de forma interactiva. El código Java que tendremos que escribir es exactamente igual que el que hemos mostrado anteriormente.

Ejercicio 1.12 En el Code Pad, escriba el código mostrado anteriormente para crear un objeto `person` e invocar sus métodos `makeVisible` y `moveRight`. Cree después algunos otros objetos y llame a sus correspondientes métodos.

Escribir estos comandos debería tener el mismo efecto que invocarlos desde el menú del objeto. Si en lugar de ello viera un mensaje de error, es que ha escrito mal el comando. Compruebe la ortografía. Observará que aunque solo haya un carácter erróneo, el comando correspondiente fallará.

Consejo Puede recuperar comandos utilizados anteriormente en el Code Pad utilizando la tecla flecha hacia arriba.

1.10 Interacción entre objetos

En la siguiente sección, vamos a trabajar con un proyecto de ejemplo distinto. Cierre el proyecto *figures* si todavía lo tiene abierto y abra el objeto denominado *house*.

Ejercicio 1.13 Abra el proyecto *house*. Cree una instancia de la clase `Picture` e invoque su método `draw`. Pruebe también los métodos `setBlackAndWhite` y `setColor`.

Ejercicio 1.14 ¿Cómo cree que dibuja una imagen la clase `Picture`?

Cinco de las clases del proyecto *house* son idénticas a las clases del proyecto *figures*. Pero ahora tenemos una clase adicional: `Picture`. Esta clase está programada para hacer exactamente lo que hemos hecho de forma manual en el Ejercicio 1.9.

En realidad, si queremos realizar una secuencia de tareas en Java, normalmente no lo haremos a mano como en el Ejercicio 1.9. En lugar de ello, crearemos una clase que lo haga por nosotros. Esta es precisamente la clase `Picture`.

La clase `Picture` está escrita de forma que, al crear una instancia, esa instancia cree dos objetos cuadrado (uno para la pared y otro para la ventana), un triángulo y un círculo; a continuación, la

Concepto:

Invocación de métodos. Los objetos pueden comunicarse entre sí **invocando** a los **métodos** de otros objetos.

instancia mueve todos esos objetos y cambia su color y su tamaño, hasta que el lienzo tiene el aspecto de la imagen mostrada en la Figura 1.8.

Lo importante aquí es que los objetos pueden crear otros objetos y pueden invocar también los métodos de otros objetos. En un programa Java normal, puede haber perfectamente cientos o miles de objetos. El usuario del programa se limita a iniciar el programa (lo que normalmente hace que se cree un primer objeto) y todos los demás objetos son creados, directa o indirectamente, por dicho objeto.

La pregunta fundamental sería entonces esta: ¿cómo escribimos una clase para dicho objeto?

1.11 Código fuente

Cada clase tiene un cierto *código fuente* asociado. El código fuente es el texto que define los detalles de la clase. En BlueJ, podemos ver el código fuente de una clase seleccionando la función *Open Editor* (Abrir editor) en el menú emergente de la clase o haciendo doble clic sobre el icono de la clase.

Concepto:

El **código fuente** de una clase determina la estructura y el comportamiento (los campos y métodos) de cada uno de los objetos de esa clase.

Ejercicio 1.15 Examine de nuevo el menú emergente de la clase `Picture`. Verá una opción denominada *Open Editor*. Selecciónela. Se abrirá un editor de textos en el que se muestra el código fuente de la clase.

El código fuente es texto escrito en el lenguaje de programación Java. Define los campos y métodos que tiene una clase y define también qué es exactamente lo que sucede cuando se invoca cada método. En el siguiente capítulo, veremos en detalle lo que contiene el código fuente de una clase y cómo está estructurado.

Una gran parte del proceso de aprendizaje del arte de la programación consiste en aprender en cómo escribir estas definiciones de clase. Para ello, aprenderemos a utilizar el lenguaje Java (aunque existen otros muchos lenguajes de programación que también pueden emplearse para escribir código).

Cuando hacemos alguna modificación en el código fuente y cerramos el editor,[3] el icono de dicha clase aparecerá con unas bandas en el diagrama. Las bandas indican que el código fuente ha sido modificado. Por ello, ahora será necesario compilar, haciendo clic en el botón *Compile*. (Lea la nota "Acerca de la compilación" para obtener más información sobre lo que sucede cuando se compila una clase.) Una vez compilada una clase, pueden volverse a crear objetos y se puede también probar la modificación realizada.

Ejercicio 1.16 En el código fuente de la clase `Picture`, localice la parte concreta que se encarga en la práctica de dibujar la imagen. Modifíquela para que el sol sea de color azul en lugar de amarillo.

Ejercicio 1.17 Añada un segundo sol a la imagen. Para ello, preste atención a las definiciones de campos situadas al principio de la clase. Allí podrá encontrar el siguiente código:

---

[3] En BlueJ, no es necesario guardar explícitamente el texto escrito en el editor antes de cerrarlo. Si se cierra el editor, el código fuente se guardará de forma automática.

```
private Square wall;
private Square window;
private Triangle roof;
private Circle sun;
```

Tendrá que añadir ahí una instrucción para definir el segundo sol. Por ejemplo:

```
private Circle sun2;
```

A continuación, escriba el código apropiado para crear ese segundo sol.

Ejercicio 1.18 *Ejercicio avanzado* (esto quiere decir que este ejercicio no puede resolverse de forma rápida. No esperamos que todos los lectores sean capaces de solucionarlo en este momento. Si consigue hacerlo, estupendo. Si no lo consigue, no se preocupe. Las cosas serán más claras a medida que progrese en la lectura. Vuelva más adelante a este ejercicio). Añada una puesta de sol a la versión de `Picture` con un único sol. Es decir, haga que el Sol vaya descendiendo lentamente. Recuerde que el círculo dispone de un método `slowMoveVertical` que puede utilizar para hacer esto.

Ejercicio 1.19 *Ejercicio avanzado.* Si ha añadido la puesta de sol al final del método `draw` (para que el Sol descienda automáticamente cuando se dibuje la imagen), modifíquelo de la forma siguiente: ahora queremos que la puesta de sol se encuentre en un método separado de forma que se pueda invocar `draw` y ver la imagen con el Sol en su posición original y luego invocar `sunset` (¡un método separado!) para hacer que el Sol se ponga.

Ejercicio 1.20 *Ejercicio avanzado.* Haga caminar a una persona hasta la casa después de la puesta del sol.

Acerca de la compilación

Cuando las personas escriben programas para computadora, normalmente utilizan un lenguaje de programación de "alto nivel" como Java. El problema que tiene esto es que una computadora no puede ejecutar directamente código fuente Java. Java fue diseñado para que fuera razonablemente fácil de leer por los seres humanos, no por las computadoras. Las computadoras trabajan internamente con una representación binaria de un código máquina, que tiene un aspecto muy distinto al de Java. El problema para nosotros es que ese código máquina parece tan complejo que no es conveniente escribir directamente con él. Es preferible escribir Java. ¿Qué podemos hacer para resolver este problema?

La solución es un programa denominado *compilador.* El compilador traduce el código Java a código máquina. Podemos escribir en Java y ejecutar el compilador –que genera el código máquina–, después de lo cual la computadora podrá leer el código máquina generado. Como resultado, cada vez que modifiquemos el código fuente deberemos primero ejecutar el compilador, antes de poder volver a utilizar la clase para crear un objeto. En caso contrario, no existiría la versión en código máquina que la computadora necesita.

1.12 Otro ejemplo

En este capítulo, hemos hablado ya de un gran número de conceptos nuevos. Para facilitar la compresión de estos conceptos, vamos a volverlos a repasar en un contexto distinto. Para ello, utilizaremos otros ejemplo. Cierre el proyecto *house* si todavía lo tiene abierto y abra el proyecto *lab-classes*.

Este proyecto es una parte simplificada de una base de datos de estudiantes diseñada para controlar a los alumnos matriculados en las clases de laboratorio e imprimir listas de esas clases.

Ejercicio 1.21 Cree un objeto de la clase `Student`. Observará que esta vez no solo se le solicita el nombre de la instancia, sino también algunos otros parámetros. Rellénelos antes de hacer clic en OK. (Recuerde que los parámetros de tipo `String` tienen que escribirse entre dobles comillas.)

1.13 Valores de retorno

Como antes, se pueden crear múltiples objetos. Y también, como antes, los objetos tienen métodos que se pueden invocar desde su menú emergente.

Ejercicio 1.22 Cree algunos objetos estudiante. Invoque el método `getName` para cada objeto. Explique lo que sucede.

Al invocar el método `getName` de la clase `Student`, podemos observar algo nuevo: los métodos pueden devolver un valor como resultado. De hecho, la signatura de cada método nos dice si el método devuelve o no un resultado y cuál es el tipo de ese resultado. La signatura de `getName` (que se muestra en el menú emergente del objeto) está definida como

Concepto:

Resultado. Los métodos pueden devolver información acerca de un objeto mediante un **valor de retorno.**

```
String getName()
```

La palabra `String` antes del nombre del método especifica el tipo de retorno. En este caso, indica que este método, al ser invocado, devuelve un resultado de tipo `String`. La signatura de `changeName` establece:

```
void changeName(String replacementName)
```

La palabra `void` indica que este método no devuelve ningún resultado.

Los métodos con valores de retorno nos permiten obtener información de un objeto mediante la invocación de un método. Esto significa que podemos emplear métodos para cambiar el estado de un objeto o para averiguar cuál es ese estado.

1.14 Objetos como parámetros

Ejercicio 1.23 Cree un objeto de clase `LabClass`. Como indica la signatura, es necesario especificar el número máximo de estudiantes en dicha clase (un entero).

Ejercicio 1.24 Invoque el método `numberOfStudents` de dicha clase. ¿Qué es lo que hace?

Ejercicio 1.25 Fíjese en la signatura del método `enrollStudent`. Observará que el tipo del parámetro esperado es `Student`. Asegúrese de tener dos o tres estudiantes y un objeto `LabClass` en el banco de

objetos y luego llame al método `enrollStudent` del objeto `LabClass` que se utiliza para matricular a un estudiante en una clase de laboratorio. Con el cursor en el campo de entrada de datos del cuadro de diálogo, haga clic en uno de los objetos estudiante; esto hará que se introduzca el nombre del objeto estudiante en el campo de parámetro del método `enrollStudent` (Figura 1.9). Haga clic en OK y con ello añadirá el estudiante a la clase de laboratorio `LabClass`. Añada algunos otros estudiantes.

Ejercicio 1.26 Invoque el método `printList` del objeto `LabClass`. Verá que en la ventana de terminal de BlueJ aparece una lista de todos los estudiantes matriculados en dicha clase (Figura 1.10).

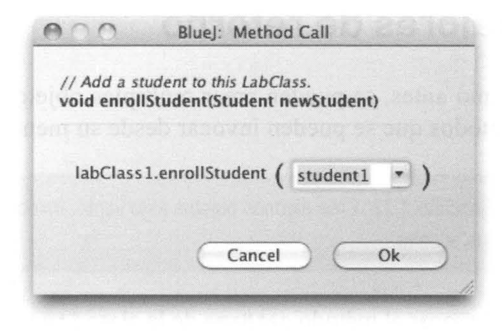


Figura 1.9
Adición de un estudiante a una clase de laboratorio `LabClass`.

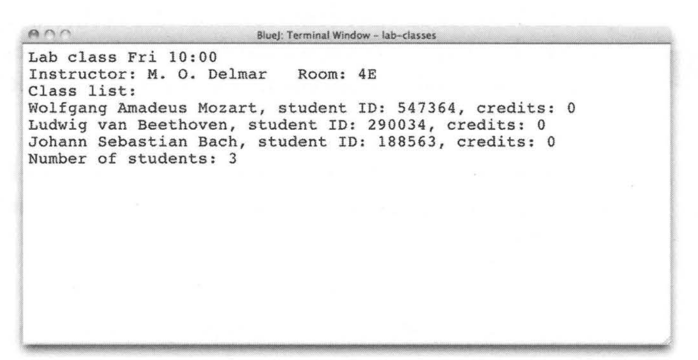


Figura 1.10
Salida del listado de matrículas de `LabClass`.

Como muestran los ejercicios, los objetos pueden pasarse como parámetros a los métodos de otros objetos. Cuando un método espera un objeto como parámetro, la signatura del método especifica como tipo de parámetro el nombre de la clase del objeto esperado.

Explore este proyecto un poco más. Trate de identificar en este contexto los conceptos expuestos en el ejemplo *figures*.

Ejercicio 1.27 Cree tres estudiantes con los siguientes detalles:

Snow White, student ID: *A00234*, credits: *24*
Lisa Simpson, student ID: *C22044*, credits: *56*
Charlie Brown, student ID: *A12003*, credits: *6*

A continuación, matricule a los tres en una clase de laboratorio y visualice el listado en pantalla.

Ejercicio 1.28 Utilice el inspector sobre un objeto `LabClass` para descubrir los campos que contiene.

Ejercicio 1.29 Defina el profesor, el aula y el horario para un laboratorio y visualice la lista en la ventana de terminal para comprobar que aparecen estos detalles.

1.15 Resumen

En este capítulo, hemos explorado los fundamentos de las clases y de los objetos. Hemos explicado el hecho de que los objetos se especifican mediante clases. Las clases representan el concepto general de las cosas, mientras que los objetos representan instancias concretas de una clase. Podemos tener múltiples objetos de cualquier clase determinada.

Los objetos disponen de métodos que utilizamos para comunicarnos con ellos. Podemos emplear un método para efectuar un cambio en el objeto o para obtener información del objeto. Los métodos pueden tener parámetros y los parámetros tienen sus correspondientes tipos. Los métodos tienen tipos de retorno, que especifican el tipo de dato que van a devolver. Si el tipo de retorno es `void`, entonces es que no devuelven nada.

Los objetos almacenan los datos en campos (que también tienen tipos). El conjunto de todos los valores de datos de un objeto se conoce como el estado del objeto.

Los objetos se crean a partir de definiciones de clases que han sido escritas en un lenguaje de programación concreto. Buena parte de la tarea de programación en Java está relacionada con cómo escribir esas definiciones de clases. Un programa Java de gran tamaño tendrá muchas clases, cada una de las cuales contará con varios métodos, que pueden llamarse unos a otros de varias formas distintas.

Para aprender a desarrollar programas Java, necesitamos aprender a escribir definiciones de clases, incluyendo sus campos y métodos, y cómo ensamblar estas clases correctamente. El resto de este libro se ocupa precisamente de estas cuestiones.

Términos introducidos en el capítulo

objeto, clase, instancia, método, signatura, parámetro, tipo, estado, código fuente, valor de retorno, compilador

Resumen de conceptos

- **objeto** Los objetos Java modelan los objetos pertenecientes a un dominio de problema.
- **clase** Los objetos se crean a partir de clases. La clase describe el tipo de objeto; los objetos representan las instanciaciones individuales de la clase.
- **método** Podemos comunicarnos con los objetos invocando métodos sobre los mismos. Si invocamos un método, los objetos normalmente llevan a cabo una acción.
- **parámetro** Los métodos pueden tener parámetros para proporcionar información adicional para una tarea.

- **signatura** A la cabecera de un método se la denomina signatura. Proporciona la información necesaria para invocar dicho método.
- **tipo** Los parámetros tienen tipos. El tipo define qué clase de valores puede tomar un parámetro.
- **instancias múltiples** Pueden crearse varios objetos similares a partir de una única clase.
- **estado** Los objetos tienen un estado. El estado se representa almacenando valores en campos.
- **invocación de métodos** Los objetos pueden comunicarse entre sí invocando a los métodos de otros objetos.
- **código fuente** El código fuente de una clase determina la estructura y el comportamiento (los campos y métodos) de cada uno de los objetos de esa clase.
- **resultado** Los métodos pueden devolver información acerca de un objeto mediante un valor de retorno.

Ejercicio 1.30 En este capítulo hemos mencionado los tipos de datos `int` y `String`. Java dispone de más tipos de datos predefinidos. Averigüe cuáles son y para qué se utilizan. Para ello, puede consultar el Apéndice B, o buscar la información correspondiente en otro libro de Java o en un manual en línea del lenguaje Java. Puede encontrar uno de tales manuales en:

```
http://download.oracle.com/javase/tutorial/java/nutsandbolts/
datatypes.html
```

Ejercicio 1.31 ¿Cuáles son los tipos de los siguientes valores?

```
0
"hello"
101
-1
true
"33"
3.1415
```

Ejercicio 1.32 ¿Qué habría que hacer para añadir un nuevo campo, por ejemplo uno denominado `name` a un objeto círculo?

Ejercicio 1.33 Escriba la signatura de un método denominado `send` que tenga un parámetro de tipo `String` y no devuelva ningún valor.

Ejercicio 1.34 Escriba la signatura de un método denominado `average` que tenga dos parámetros, ambos de tipo `int` y devuelva un valor de tipo `int`.

Ejercicio 1.35 Mire el libro que está leyendo en este momento. ¿Es un objeto o una clase? Si es una clase, enumere algunos objetos. Si es un objeto, indique cuál es su clase.

Ejercicio 1.36 ¿Puede un objeto tener varias clases distintas? Explique su respuesta.

CAPÍTULO

2 Definiciones de clases

Principales conceptos explicados en el capítulo:

- campos
- constructores
- parámetros
- métodos (selector, mutador)
- instrucciones de asignación y condicionales

Estructuras Java explicadas en este capítulo:

campo, constructor, comentario, parámetro, asignación (=), bloque, instrucción return, `void`, operadores de asignación compuestos (+=, −=), instrucción if

En este capítulo, vamos a echar un primer vistazo detallado al código fuente de una clase. Explicaremos los elementos básicos de las definiciones de clase: *campos*, *constructores* y *métodos*. Los métodos contienen instrucciones, e inicialmente nos limitaremos a examinar métodos que solo contendrán instrucciones aritméticas simples e instrucciones de impresión. Posteriormente, presentaremos las *instrucciones condicionales* que permiten elegir entre diferentes acciones que llevar a cabo dentro de los métodos.

Comenzaremos examinando un nuevo proyecto con un alto grado de detalle. Este proyecto representa una implementación simple de una máquina expendedora de billetes automatizada. Cuando comencemos presentando las características más básicas de las clases, veremos enseguida que esta implementación tiene numerosas carencias. Por ello, procederemos a describir una versión más sofisticada de la máquina expendedora que nos permitirá conseguir una mejora significativa. Finalmente, para reforzar los conceptos presentados en el capítulo, examinaremos los detalles internos del ejemplo *lab-classes* que hemos visto en el Capítulo 1.

2.1 Máquinas expendedoras

En las estaciones de tren a menudo suele haber máquinas expendedoras que imprimen un billete cuando un cliente inserta la cantidad correcta de dinero. En este capítulo vamos a definir una clase que permite modelar algo parecido a estas máquinas expendedoras. Puesto que nuestro objetivo es examinar unas primeras clases Java de ejemplo, comenzaremos manteniendo un alto grado de simplicidad en nuestra simulación. Esto nos dará la

oportunidad de plantearnos algunas cuestiones acerca de cómo difieren estos modelos con respecto a sus correspondientes versiones del mundo real y cómo podemos modificar nuestras clases para hacer que los objetos que se crean a partir de las mismas se parezcan más a sus contrapartidas reales.

Nuestras máquinas expendedoras funcionan de la forma siguiente: los clientes "insertan" dinero en ellas y luego solicitan que se imprima un billete. Cada máquina lleva la cuenta del total de dinero acumulado desde que su puesta en funcionamiento. En la vida real, a menudo sucede que las máquinas expendedoras ofrecen distintos tipos de billetes, de entre los cuales el cliente selecciona el que quiere. Nuestra máquina simplificada imprimirá billetes de un único precio. Resulta significativamente más complicado programar una clase que sea capaz de emitir billetes de diferentes valores, que otra que solo admita un único valor. Por otro lado, con la programación orientada a objetos, es muy sencillo crear múltiples instancias de la clase, cada una con su propio precio asociado, si queremos satisfacer la necesidad de diferentes tipos de billetes.

2.1.1 Comportamiento de una máquina expendedora simple

Concepto:

Creación de objetos Algunos objetos no pueden construirse a menos que proporcionemos información adicional.

Abra el proyecto *naive-ticket-machine* en BlueJ. Este proyecto solo tiene una clase –`TicketMachine`– que vamos a poder explorar de forma similar a los ejemplos vistos en el Capítulo 1. Cuando se crea una instancia de `TicketMachine`, se nos pide que suministremos un número, que se corresponde con el precio de los billetes emitidos por esa máquina concreta. Consideraremos que el precio es una cierta cantidad de céntimos, por lo que un ejemplo de valor apropiado con el que trabajar sería un número entero positivo como 500.

Ejercicio 2.1 Cree un objeto `TicketMachine` en el banco de objetos y examine sus métodos. Debería ver los siguientes: `getBalance`, `getPrice`, `insertMoney` y `printTicket`. Pruebe el método `getPrice`, que indica el precio del billete. Debería ver un valor de retorno que contiene el precio de los billetes que se configuró cuando se creó este objeto. Utilice el método `insertMoney` para simular la inserción de una cierta cantidad en la máquina. La máquina almacena como balance la cantidad de dinero introducida. Utilice `getBalance` para comprobar que la máquina ha anotado de manera precisa la cantidad de dinero que acabamos de insertar. Se pueden introducir varias cantidades separadas de dinero en la máquina, de la misma forma que se meterían varias monedas o billetes en una máquina real. Pruebe a insertar la cantidad exacta requerida para obtener un billete y emplee `getBalance` para verificar que el balance se ha incrementado adecuadamente. Puesto que se trata de una máquina simple, esta no emitirá un billete automáticamente, así que cuando haya insertado la cantidad de dinero suficiente, deberá llamar al método `printTicket`. Deberá entonces imprimir un facsímil de billete en la ventana de terminal de BlueJ.

Ejercicio 2.2 ¿Qué valor se devuelve si consultamos el balance de la máquina después de haber impreso el billete?

Ejercicio 2.3 Experimente insertando diferentes cantidades de dinero antes de imprimir los billetes. ¿Observa algo extraño en el comportamiento de la máquina? ¿Qué sucede si inserta demasiado dinero en la máquina: le devuelve la máquina dinero? ¿Qué sucede si no mete el dinero suficiente y después intenta imprimir el billete?

Ejercicio 2.4 Trate de comprender a fondo el comportamiento de la máquina expendedora interactuando con ella en el banco de objetos antes de comenzar a ver, en la siguiente sección, cómo está implementada la clase `TicketMachine`.

Ejercicio 2.5 Cree otra máquina expendedora para billetes con un precio distinto; recuerde que deberá especificar dicho valor en el momento de crear el objeto máquina. Adquiera un billete en dicha máquina. ¿Tiene un aspecto distinto el billete impreso, comparado con los que imprime la primera máquina?

2.2 Examen de la definición de una clase

Los ejercicios al final de la sección anterior revelan que los objetos `TicketMachine` solo se comportan realmente de la forma que esperamos si introducimos exactamente la cantidad correcta de dinero definida como precio del billete. Cuando exploremos los detalles internos de la clase en esta sección, entenderemos por qué sucede esto así.

Examine el código fuente de la clase `TicketMachine` haciendo doble clic en su icono en el diagrama de clases dentro de BlueJ. Debería tener un aspecto similar al de la Figura 2.1.

Figura 2.1
La ventana del editor de BlueJ.

TicketMachine

Compile | Undo | Cut | Copy | Paste | Find... | Close | Source Code

```
 * @author David J. Barnes and Michael Kolling
 * @version 2008.03.30
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     * Note that the price must be greater than zero, and there
     * are no checks to ensure this.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * Return the price of a ticket.
     */
    public int getPrice()
```

saved

El texto completo de la clase se muestra en el Código 2.1 (en el que podrá encontrar los comentarios traducidos al español). Examinando el texto de la definición de la clase parte por parte, podemos entender mejor algunos de los conceptos de la orientación a objetos de los que hemos hablado en el Capítulo 1. Esta definición de clase contiene muchas de las características de Java con las que nos toparemos una y otra vez, por lo que merece la pena estudiarlo cuidadosamente.

Código 2.1
La clase `TicketMachine`.

```
/**
 * TicketMachine modela una máquina expendedora simple que emite
 * billetes de un único precio.
 * El precio de un billete se especifica mediante el constructor.
```

Código 2.1
(continuación)
La clase `TicketMachine`

```
 * Es una máquina poco inteligente, en el sentido de que confía en
 * que los usuarios inserten el suficiente dinero antes de intentar
 * imprimir un billete. También supone que el usuario introduce
 * cantidades que tengan sentido.
 *
 * @author David J. Barnes y Michael Kölling
 * @version 2011.07.31
 */
public class TicketMachine
{
    // Precio de un billete en esta máquina.
    private int price;
    // Cantidad de dinero insertada hasta el momento por el cliente.
    private int balance;
    // Cantidad total de dinero recaudado por la máquina.
    private int total;

    /**
     * Crear una máquina que emita billetes del precio indicado.
     * Observe que el precio tiene que ser mayor que cero y que no
     * se efectúa ninguna comprobación para verificar esto.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * Devuelve el precio de un billete.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Devuelve la cantidad de dinero ya insertada para el
     * siguiente billete.
     */
    public int getBalance()
    {
        return balance;
    }

    /**
     * Recibe una cierta cantidad de dinero de un cliente.
     */
```

Código 2.1
(continuación)
La clase `TicketMachine`

```
    public void insertMoney(int amount)
    {
        balance = balance + amount;
    }

    /**
     * Imprimir un billete.
     * Actualizar el total recaudado y poner el balance a cero.
     */
    public void printTicket()
    {
        // Simular la impresión de un billete.
        System.out.println("##################");
        System.out.println("# The BlueJ Line");
        System.out.println("# Ticket");
        System.out.println("# " + price + " cents.");
        System.out.println("##################");
        System.out.println();

        // Actualizar el total recaudado con el balance.
        total = total + balance;
        // Poner el balance a cero.
        balance = 0;
    }
}
```

2.3 La cabecera de la clase

El texto de una clase puede dividirse en dos partes principales: un envoltorio exterior que simplemente da nombre a la clase y una parte interna, mucho más larga, que se encarga de realizar todo el trabajo. En este caso, ese envoltorio exterior tiene la siguiente apariencia:

```
public class TicketMachine
{
    Se omite la parte interna de la clase.
}
```

El envoltorio exterior de las diferentes clases se parece bastante. Ese envoltorio exterior contiene la cabecera de la clase, cuyo propósito principal es el de proporcionar a la clase un nombre. De acuerdo con un convenio ampliamente aceptado, los nombres de las clases comienzan siempre con una letra mayúscula. Siempre que se emplee de manera constante, este convenio permite distinguir fácilmente los nombres de las clases de otros tipos de nombres, como los nombres de variables y los nombres de métodos, que describiremos más adelante.

Ejercicio 2.6 Escriba lo que crea que deberían ser los envoltorios externos de las clases `Student` y `LabClass`; no se preocupe por la parte interna.

Ejercicio 2.7 ¿Tiene alguna importancia si escribimos

```
public class TicketMachine
```

o

```
class public TicketMachine
```

en el envoltorio externo de una clase? Edite el código fuente de la clase `TicketMachine` para realizar esa modificación y luego cierre la ventana del editor. ¿Observa algún cambio en el diagrama de clases?

¿Qué mensaje de error se obtiene al pulsar ahora el botón *Compile*? ¿Cree que este mensaje explica claramente qué es lo que está mal?

Escriba otra vez la clase con su formato original y asegúrese de que con esto desaparece el error al compilar la clase de nuevo.

Ejercicio 2.8 Compruebe si es posible o no eliminar la palabra `public` del envoltorio externo de la clase `TicketMachine`.

Ejercicio 2.9 Vuelva a incluir la palabra `public` y luego compruebe si es posible eliminar la palabra `class` tratando de compilar de nuevo. Asegúrese antes de continuar de volver a incluir ambas palabras tal y como estaban escritas originalmente.

2.3.1 Palabras clave

Las palabras "public" y "class" son parte del lenguaje Java, mientras que "TicketMachine" no lo es –la persona que ha escrito esa clase ha elegido ese nombre concreto. A las palabras como "public" y "class" las denominamos *palabras clave* o *palabras reservadas* –ambos términos se utilizan frecuentemente y de manera intercambiable. Existen alrededor de 50 palabras de estas en Java, y pronto se acostumbrará a reconocer la mayor parte de ellas. Un aspecto que conviene recordar es que las palabras clave Java nunca contienen letras mayúsculas, mientras que las palabras que elegimos como programadores (como por ejemplo "TicketMachine") son a menudo una mezcla de letras mayúsculas y minúsculas.

2.4 Campos, constructores y métodos

La parte interna de la clase es donde definimos los *campos, constructores* y *métodos* que proporcionan a los objetos de dicha clase sus propias características y sus propios comportamientos particulares. Podemos resumir las características esenciales de estos tres componentes de una clase de la forma siguiente:

- Los campos almacenan datos de manera persistente dentro de un objeto.
- Los constructores son responsables de garantizar que un objeto se configure apropiadamente en el momento de crearlo por primera vez.
- Los métodos implementan el comportamiento de un objeto; proporcionan su funcionalidad.

En Java, hay muy pocas reglas acerca del orden en que se deben definir los campos, constructores y métodos dentro de una clase. En la clase `TicketMachine`, hemos decidido enumerar primero los campos, luego los constructores y por último los métodos (Código 2.2).

Este es el orden que seguiremos en todos nuestros ejemplos. Otros autores prefieren adoptar estilos distintos, y se trata fundamentalmente de una cuestión de gusto personal. Nuestro estilo no es necesariamente mejor que el de otras personas. Sin embargo, lo que sí es importante es elegir un cierto estilo y luego utilizarlo de forma constante, porque de este modo, las clases que programemos serán más fáciles de leer y de entender.

Código 2.2
Nuestra ordenación de campos, constructores y métodos.

```
public class NombreClase
{
    Campos
    Constructores
    Métodos
}
```

Ejercicio 2.10 Teniendo en cuenta nuestros experimentos anteriores con los objetos máquina expendedora en BlueJ, probablemente recuerde los nombres de algunos de los métodos, como por ejemplo `printTicket`. Observe la definición de la clase en el Código 2.1 y utilice esos conocimientos junto con la información adicional acerca de la ordenación que acabamos de comentar para hacer una lista de los nombres de los campos, constructores y métodos de la clase `TicketMachine`. *Pista:* esta clase tiene un único constructor.

Ejercicio 2.11 ¿Cuáles son las dos características del constructor que hace que tenga un aspecto significativamente distinto de los métodos de la clase?

2.4.1 Campos

Concepto:

Los **campos** almacenan datos que un objeto tiene que utilizar. Los campos se conocen también con el nombre de variables de instancia.

Los campos almacenan datos de manera persistente dentro de un objeto. La clase `TicketMachine` tiene tres campos: `price`, `balance` y `total`. Los campos también se conocen con el nombre de *variables de instancia,* porque la palabra *variable* se utiliza como término general para todos aquellos elementos que permiten almacenar datos en un programa. Hemos definido los campos justo al principio de la definición de la clase (Código 2.3). Todas estas variables están asociadas con elementos monetarios con los que un objeto máquina expendedora tiene que tratar:

- `price` almacena el precio fijado para un billete;
- `balance` almacena la cantidad de dinero insertada en la máquina por un usuario, antes de pedir que se imprima un billete;
- `total` almacena la cantidad total de dinero insertada en la máquina por todos los usuarios desde que se construyó el objeto máquina (excluyendo el balance actual). La idea es que cuando se imprime un billete, el dinero reflejado por el balance se transfiera al total.

Código 2.3
Los campos de la clase `TicketMachine`.

```
public class TicketMachine
{
    private int price;
    private int balance;
    private int total;
    Se omiten el constructor y los métodos.
}
```

Los campos son pequeñas cantidades de espacio dentro de un objeto que pueden emplearse para almacenar datos de manera persistente. Todos los objetos tendrán espacio para cada campo declarado en su clase. La Figura 2.2 muestra una diagrama de un objeto máquina expendedora con sus tres casos. Todavía no hemos asignado ningún valor a esos campos; una vez que lo hagamos, podemos escribir cada valor en el recuadro que representa al campo correspondiente. La notación es similar a la que se utiliza en BlueJ para mostrar objetos en el banco de objetos, salvo porque aquí mostramos algo más de detalle. En BlueJ, por razones de espacio, los campos no se muestran dentro del icono del objeto. Sin embargo, podemos verlos abriendo una ventana del inspector (Sección 1.5).

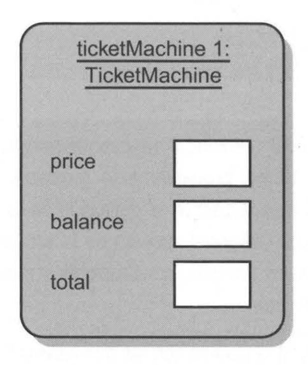


Figura 2.2
Un objeto de la clase `TicketMachine`.

Concepto:

Los **comentarios** se insertan en el código fuente de una clase para proporcionar explicaciones a los lectores humanos. No tienen ningún efecto sobre la funcionalidad de la clase.

Cada campo dispone de su propia declaración en el código fuente. Dentro de la definición completa de la clase, en la línea situada encima de cada campo, hemos añadido una única línea de texto, un *comentario*, para facilitar la tarea a las personas que quieran leer la definición de la clase:

```
// Precio del billete de esta máquina.
private int price;
```

Los comentarios de una única línea van precedidos por los dos caracteres "//", que se escriben sin ningún espacio entre ellos. Los comentarios más detallados, que abarcan a menudo varias líneas, suelen escribirse en forma de comentarios multilínea, que comienzan con la pareja de caracteres "/*" y terminan con la pareja "*/". Hay un buen ejemplo justo antes de la cabecera de la clase en el Código 2.1.

Las definiciones de los tres campos son bastante similares:

- Todas las definiciones indican que se trata de campos *privados* del objeto; diremos algo más acerca del significado de esto en el Capítulo 5, pero por el momento dejemos claro simplemente que los campos siempre se definen como privados (`private`).
- Los tres campos son de tipo `int` –`int` es otra palabra clave y representa el tipo de datos entero. Indica que cada campo puede almacenar un único valor entero, lo cual es razonable, ya que los queremos para almacenar números que representen cantidades de dinero en céntimos.

Como los campos pueden almacenar valores que varíen con el tiempo, se les conoce también con el nombre de *variables*. En caso necesario, el valor almacenado en un campo puede modificarse con respecto a su valor inicial. Por ejemplo, a medida que se inserta más dinero en una máquina expendedora, necesitaremos cambiar el valor almacenado en el campo `balance`.

Es bastante común tener algunos campos cuyos valores cambien a menudo, como `balance` y `total`, y otros que cambien rara vez o no cambien en absoluto como `price`. El que el valor de `price` no varíe después de haber configurado ese campo no cambia el hecho de que se le continúe llamando variable. En las siguiente secciones, veremos algunos otros tipos de variables distintas de los campos, pero todas ellas comparten el mismo propósito fundamental: almacenar datos.

Los campos `price`, `balance` y `total` son todos los elementos de datos que un objeto máquina expendedora necesita para cumplir la función de recibir dinero de un cliente, imprimir billetes y mantener el total de dinero que se ha introducido en la máquina. En las siguientes secciones, veremos cómo el constructor y los métodos utilizan esos campos para implementar el comportamiento de estas máquinas expendedoras tan sencillas.

Ejercicio 2.12 ¿Cuál cree que es el *tipo* de cada uno de los siguientes campos?

```
private int count;
private Student representative;
private Server host;
```

Ejercicio 2.13 ¿Cuáles son los *nombres* de los siguientes campos?

```
private boolean alive;
private Person tutor;
private Game game;
```

Ejercicio 2.14 Con lo que sabe acerca de los convenios de denominación de las clases, ¿cuáles de los nombres de tipo de los Ejercicios 2.12 y 2.13 diría que son nombres de clases?

Ejercicio 2.15 En la siguiente declaración de campo de la clase `TicketMachine`

```
private int price;
```

¿importa el orden en el que aparecen las tres palabras? Edite la clase `TicketMachine` para probar diferentes ordenaciones. Después de cada modificación, cierre el editor. ¿Cree que la apariencia del diagrama de clases después de cada cambio proporciona alguna indicación sobre si son posibles otras ordenaciones? Compruebe si es así haciendo clic en el botón *Compile* para ver si aparece un mensaje de error.

¡ Asegúrese de restaurar la versión original después de realizar sus experimentos!

Ejercicio 2.16 ¿Es siempre necesario incluir un punto y coma al final de la declaración de un campo? Una vez más, experimente utilizando el editor. La regla que aprenderá con ello es importante, así que asegúrese de recordarla.

Ejercicio 2.17 Escriba la declaración completa de un campo de tipo `int` cuyo nombre sea `status`.

A partir de las definiciones de campos que hemos visto hasta ahora, podemos comenzar a deducir un cierto patrón que será de aplicación cada vez que definamos una variable de campo dentro de una clase:

- Normalmente, comienzan con la palabra reservada `private`.
- Incluyen un nombre de tipo (como por ejemplo `int`, `String`, `Person`, etc.)
- Incluyen un nombre elegido por el usuario para la variable de campo.

- Terminan con un punto y coma.

Recordar este patrón le será de gran ayuda cuando escriba sus propias clases.

De hecho, cuando examinemos detalladamente el código fuente de diferentes clases, veremos cómo emergen patrones como este una y otra vez. Parte del proceso de aprender a programar implica buscar dichos patrones y luego utilizarlos en nuestros propios programas. Esa es una de las razones por las que el estudio detallado de código fuente resulta tan útil en esta etapa.

2.4.2 Constructores

Concepto:

Los **constructores** permiten configurar cada objeto apropiadamente en el momento de crearlo por primera vez.

Los constructores tienen un papel especial que cumplir. Son responsables de garantizar que cada objeto se configure adecuadamente en el momento de crearlo por vez primera. En otras palabras, garantizan que cada objeto esté listo para ser utilizado inmediatamente después de su creación. Este proceso de construcción también se denomina *inicialización.*

En algunos aspectos, el constructor puede asemejarse a una comadrona: es responsable de garantizar que el nuevo objeto comience su vida apropiadamente. Una vez creado un objeto, el constructor no juega ningún papel ulterior en la vida del objeto y no puede invocarse para ese objeto. El Código 2.4 muestra el constructor de la clase `TicketMachine`.

Una de las características distintivas de los constructores es que tienen el mismo nombre de la clase en la se encuentran definidos –`TicketMachine` en este caso. El nombre del constructor sigue inmediatamente a la palabra `public`, sin ningún otro elemento entre ellos.[1]

Cabe esperar que exista una estrecha conexión entre lo que sucede en el cuerpo de un constructor y en los campos de la clase. Esto se debe a que uno de los papeles principales del constructor es el de inicializar los campos. Es posible con algunos campos, como por ejemplo

Código 2.4
El constructor de la clase `TicketMachine`.

```
public class TicketMachine
{
    Campos omitidos.

    /**
     * Crear una máquina que emita billetes del precio indicado.
     * Observe que el precio tiene que ser mayor que cero y que no
     * se efectúa ninguna comprobación para verificar esto.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    Métodos omitidos.
}
```

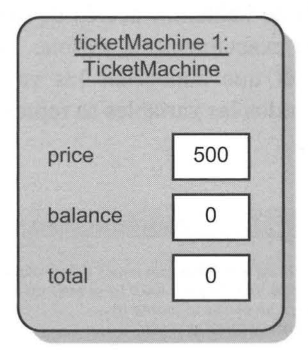
[1] Aunque esta descripción es una ligera simplificación de la regla completa correspondiente de Java, encaja con la regla general que utilizaremos en la mayor parte del código incluido en este libro.

`balance` y `total`, establecer valores iniciales adecuados, asignando a esos campos un número constante –en este caso, cero. Con otros campos, como el correspondiente al precio del billete, no es tan simple, ya que no conoceremos el precio de los billetes para una máquina concreta hasta que esa máquina sea construida. Recuerde que queremos poder crear múltiples objetos máquina para vender billetes de diferentes precios, así que no hay ningún posible precio inicial que sea siempre correcto. Si ha estado experimentando con la creación de objetos `TicketMachine` en BlueJ, ya sabrá que es preciso proporcionar el coste de los billetes cada vez que se crea una nueva máquina expendedora. Un aspecto importante que hay que recalcar aquí es que el precio de un billete se determina inicialmente *de manera externa*, teniendo que *pasarse* después al constructor. En BlueJ, somos nosotros los que decidimos cuál es ese valor y lo especificamos en un cuadro de diálogo. Parte de la tarea del constructor consiste en recibir ese valor y almacenarlo en el campo `price` de la máquina expendedora recién creada, con el fin de que la máquina pueda recordar cuál era ese valor, sin que nosotros tengamos que volver a proporcionárselo.

Podemos ver a partir de esto que una de las funciones más importantes de un campo consiste en recordar la información externa pasada al objeto, con el fin de que esa información esté disponible para el objeto a lo largo de toda su vida. Los campos proporcionan, por tanto, un lugar para almacenar datos de larga duración (es decir, persistentes).

La Figura 2.3 muestra un objeto máquina expendedora después de haberse ejecutado el constructor. Como vemos, ahora se han asignado valores a los campos. A partir de este diagrama, se puede deducir que la máquina expendedora se creó pasándole el valor 500 como precio del billete.

Figura 2.3
Un objeto `TicketMachine` después de la inicialización (creado para billetes de 500 céntimos).



En la siguiente sección, explicaremos cómo reciben los objetos los valores procedentes del exterior.

Nota En Java, todos los campos se inicializan automáticamente con un cierto valor predeterminado cuando no se inicializan de manera explícita. Para los campos enteros, este valor predeterminado es cero. Por tanto, no sería estrictamente necesario que nos preocupáramos de definir los campos de balance y total con el valor cero, ya que el valor predeterminado nos permitirá obtener el mismo resultado, Sin embargo, es preferible escribir de todos modos las asignaciones explícitamente. No tiene ninguna desventaja y permite documentar mejor qué es lo que está sucediendo en realidad. No hace falta confiar en que la persona que lea el código de la clase sepa cuál es el valor predeterminado, y además dejaremos documentado que realmente queremos que ese valor sea cero evitando la duda de si no nos habremos olvidado de inicializarlo.

2.5 Parámetros: recepción de datos

Los constructores y los métodos desempeñan papeles muy distintos en la vida de un objeto, pero la forma en que ambos reciben valores desde el exterior es la misma: a través de *parámetros*. Recuerde que ya nos hemos topado brevemente con los parámetros en el Capítulo 1 (Sección 1.4). Los parámetros son otro tipo de variable, igual que los campos, por lo que se utilizan para almacenar datos. Los parámetros son variables que se definen en la cabecera de un constructor o de un método:

```
public TicketMachine(int cost)
```

Este constructor tiene un único parámetro, `cost`, que es de tipo `int` –el mismo tipo que el campo `price`, que es el campo que queremos configurar con este parámetro. Los parámetros se emplean como una especie de mensajeros temporales, que transportan datos que tienen su origen fuera del constructor o método y que hacen que esos datos estén disponibles en el interior del constructor o método.

La Figura 2.4 ilustra cómo se pasan los valores mediante parámetros. En este caso, un usuario de BlueJ introduce el valor externo en el cuadro de diálogo en el momento de crear una nueva máquina expendedora (mostrada a la izquierda). Luego dicho valor se copia en el parámetro `cost` del constructor de la nueva máquina. Esto se ilustra mediante la flecha etiquetada (A). El recuadro en el objeto `TicketMachine` de la Figura 2.4, etiquetado como "TicketMachine (constructor)", representa un espacio adicional para el objeto que solo se crea cuando el constructor se ejecuta. Lo denominaremos *espacio del constructor* del objeto (o *espacio del método* cuando hablemos acerca de métodos y no de constructores), ya que en aquel caso la situación es exactamente la misma. El espacio del constructor se utiliza para proporcionar espacio en el que almacenar los valores de los parámetros del constructor. En nuestros diagramas, todos las variables se representan mediante recuadros blancos.

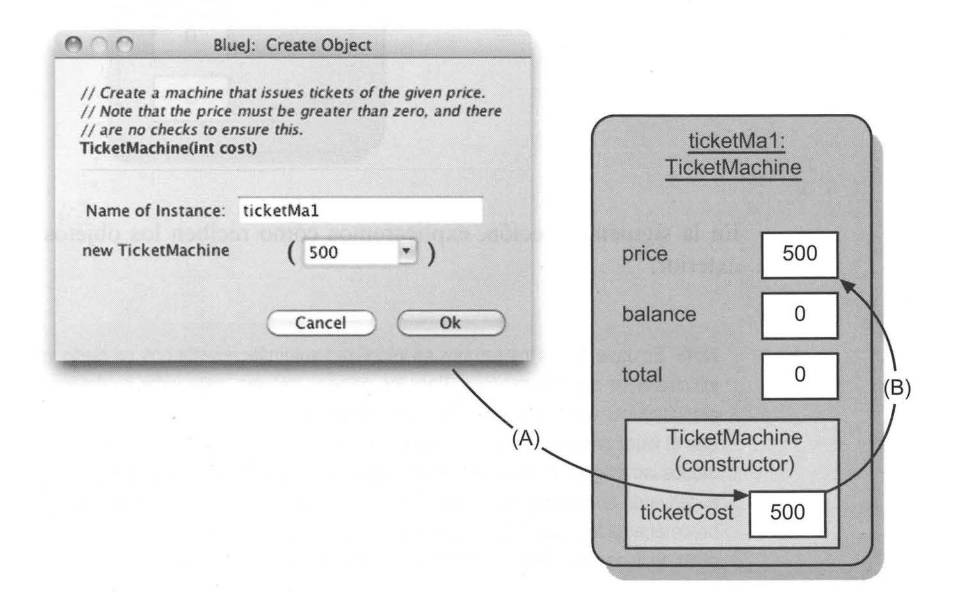


Figura 2.4
Paso de parámetros (A) y asignación (B).

Concepto:

El **ámbito** de una variable define la sección del código fuente desde la que se puede acceder a esa variable.

Concepto:

El **tiempo de vida** de una variable describe durante cuánto tiempo continúa existiendo la variable antes de ser destruida.

Distinguiremos entre los *nombres* de los parámetros dentro de un constructor o método y los *valores* externos de los parámetros, denominando a los nombres *parámetros formales* y a los valores *parámetros reales*. Así, `cost` es un parámetro formal, mientras que un valor suministrado por el usuario, como por ejemplo 500, es un parámetro real.

Un parámetro formal solo está disponible para un objeto dentro del cuerpo de un constructor o método que lo declare. Decimos que el *ámbito* de un parámetro está restringido al cuerpo del constructor o método en el que se declara. Por el contrario, el ámbito de un campo es todo el conjunto de la definición de la clase –puede accederse a él desde cualquier punto de la misma clase. Esta es una diferencia muy importante entre estos dos tipos de variables.

Un concepto relacionado con el ámbito de las variables es el *tiempo de vida* de las mismas. El tiempo de vida de un parámetro está limitado a una única llamada a un constructor o método. Cuando se invoca un constructor o método, se crea el espacio adicional para las variables de parámetro y los valores externos se copian en dicho espacio. Una vez que la llamada ha completado su tarea, los parámetros formales desaparecen y los valores que contenían se pierden. En otras palabras, cuando el constructor ha terminado de ejecutarse, se elimina todo el espacio del constructor (véase la Figura 2.4), junto con las variables de parámetro contenidas dentro del mismo.

Por el contrario, el tiempo de vida de un campo coincide con el del objeto al que pertenece. Cuando se crea un objeto, se crean también todos los campos del mismo, y esos campos persisten mientras dure el tiempo de vida del objeto. De aquí se deduce que, si queremos recordar el coste de los billetes almacenado en el parámetro `cost`, deberemos almacenar dicho valor en algún lugar persistente; es decir, en el campo `price`.

Al igual que cabía esperar que existiera una estrecha conexión entre un constructor y los campos de una clase, también cabe esperar que exista una estrecha conexión entre los parámetros del constructor y los campos, porque a menudo se necesitarán valores externos para configurar los valores iniciales de uno o más de esos campos. Cuando esto sea así, los tipos de los parámetros se asemejerán estrechamente a los tipos de los campos correspondientes.

Ejercicio 2.18 ¿A qué clase pertenece el siguiente constructor?

```
public Student(String name)
```

Ejercicio 2.19 ¿Cuántos parámetros tiene el siguiente constructor y cuáles son sus tipos?

```
public Book(String title, double price)
```

Ejercicio 2.20 ¿Podría adivinar de qué tipo serán algunos de los campos de la clase `Book` a partir de la definición de los parámetros de su constructor? ¿Podemos hacer alguna suposición acerca de los nombres de esos campos?

2.5.1 Elección de los nombres de variable

Una de las cosas que puede que haya observado es que los nombres de variable que utilizamos para los campos y los parámetros tienen una estrecha conexión con el propósito de la variable. Nombres como `price`, `cost`, `title` y `alive` nos proporcionan indicaciones útiles acerca de la información que se está almacenando en esas variables. A su vez, esto hace que sea más fácil comprender lo que pasa en el programa. Dado que tenemos una

gran libertad a la hora de seleccionar los nombres de las variables, es conveniente elegir nombres que proporcionen algo de información al lector, en lugar de elegir combinaciones arbitrarias e ininteligibles de letras y números.

2.6 Asignación

Concepto:

Las **instrucciones de asignación** almacenan el valor representado por el lado derecho de la instrucción en la variable especificada a la izquierda.

En la sección anterior, hemos observado que nos hace falta tomar el valor de corta duración contenido en una variable de parámetro y almacenarlo en algún otro lugar más permanente –una variable de campo. Para hacer esto, el cuerpo del constructor contiene la siguiente *instrucción de asignación:*

```
price = cost;
```

Las instrucciones de asignación se emplean con enorme frecuencia en la programación, como medio de almacenar un valor en una variable. Pueden reconocerse por la presencia de un operador de asignación, como "=" en el ejemplo anterior. Las instrucciones de asignación funcionan tomando el valor que aparece en el lado derecho del operador y copiando dicho valor en la variable especificada en el lado izquierdo. Esto se ilustra en la Figura 2.4 mediante la flecha etiquetada como (B). El lado derecho se denomina *expresión.* En su forma más general, las expresiones son elementos que calculan un valor, pero en este caso la expresión consiste únicamente en una sola variable, cuyo valor se copia en la variable `price`. Veremos ejemplos de expresiones más complicadas posteriormente en el capítulo.

Una regla relativa a las instrucciones de asignación es que el tipo de la expresión del lado derecho debe corresponderse con el tipo de la variable a la que se asigna. Hasta el momento nos hemos encontrado con tres tipos diferentes de uso común: `int`, `String` y (muy brevemente) `boolean`. Esta regla implica que no podemos almacenar, por ejemplo, una expresión de tipo `int` en una variable de tipo `String`. Esta misma regla también se aplica entre parámetros formales y parámetros reales: el tipo de una expresión de parámetro real debe corresponderse con el tipo de la variable que actúa como parámetro formal. Por ahora, limitémonos a decir que los tipos de ambas deben ser iguales, aunque en posteriores capítulos veremos que la realidad es un poco más complicada.

Ejercicio 2.21 Suponga que la clase `Pet`, utilizada para modelar mascotas, tiene un campo denominado `name` que es de tipo `String`. Escriba una instrucción de asignación en el cuerpo del siguiente constructor para inicializar el campo `name` con el valor del parámetro del constructor.

```
public Pet(String petsName)
{
}
```

Ejercicio 2.22 *Ejercicio avanzado.* La siguiente operación de creación de un objeto hará que se invoque el constructor de la clase `Date`, utilizada para modelar fechas. ¿Puede escribir la cabecera del constructor?

```
new Date("March", 23, 1861)
```

Trate de proporcionar nombres significativos a los parámetros.

2.7 Métodos

La clase `TicketMachine` tiene cuatro métodos: `getPrice`, `getBalance`, `insertMoney` y `printTicket`. Comenzaremos nuestro examen del código fuente de los métodos analizando `getPrice` (Código 2.5).

Código 2.5
El método getPrice.

```
public class TicketMachine
{
    Campos omitidos.
    Constructor omitido.

    /**
     * Devuelve el precio de un billete.
     */
    public int getPrice()
    {
        return price;
    }
    Restantes métodos omitidos.
}
```

Concepto:

Los **métodos** están compuestos de dos partes: una cabecera y un cuerpo.

Los métodos tienen dos partes: una *cabecera* y un *cuerpo*. He aquí la cabecera del método `getPrice`, precedida por un comentario descriptivo:

```
/**
 * Devuelve el precio de un billete.
 */
public int getPrice()
```

Es importante distinguir entre las cabeceras de los métodos y las declaraciones de los campos, porque pueden parecer bastante similares. Podemos ver que `getPrice` es un método y no un campo porque las cabeceras de los métodos siempre incluyen una pareja de paréntesis –"(" y ")"– y no incluyen punto y coma al final de la cabecera.

El cuerpo del método es el resto del método, es decir, el código situado después de la cabecera. Siempre se encierra entre un par de llaves: "{" y "}". Los cuerpos de los métodos contienen las *declaraciones* y las *instrucciones* que definen lo que hace un objeto cuando se invoca ese método. Las declaraciones se utilizan para crear espacio adicional de variables temporales, mientras que las instrucciones describen las acciones del método. En `getPrice`, el cuerpo del método contiene una única instrucción, pero pronto veremos ejemplos en los que el cuerpo del método está compuesto por múltiples líneas tanto de declaraciones como de instrucciones.

Cualquier conjunto de declaraciones e instrucciones situado entre una pareja de llaves se conoce con el nombre de *bloque*. Por tanto, el cuerpo de la clase `TicketMachine` y los cuerpos del constructor y de todos los métodos de la clase son bloques.

Existen al menos dos diferencias significativas entre las cabeceras del constructor de `TicketMachine` y del método `getPrice`:

```
public TicketMachine(int cost)
public int getPrice()
```

- El método tiene un *tipo de retorno* `int`, mientras que el constructor no tiene ningún tipo de retorno. El tipo de retorno se escribe justo delante del nombre del método. Esta es una diferencia que se aplica en todos los casos.
- El constructor tiene un único parámetro formal, `cost`, mientras que el método no tiene ninguno –solo una pareja de paréntesis vacíos. Esta es una diferencia que se aplica en este caso concreto.

En Java, una regla que se aplica de manera general es que los constructores no pueden tener un tipo de retorno. Por otro lado, tanto los constructores como los métodos pueden tener cualquier número de parámetros formales, incluyendo ninguno.

Dentro del cuerpo de `getPrice` hay una única instrucción:

```
return price;
```

Esta instrucción se denomina *instrucción de retorno (return).* Es responsable de devolver un valor entero que se corresponda con el tipo de retorno `int` especificado en la cabecera del método. Cuando un método contiene una instrucción de retorno, será siempre la instrucción final de dicho método, porque una vez que se ejecute dicha instrucción de retorno no se podrá ejecutar ninguna instrucción adicional en ese método.

Los tipos de retorno y las instrucciones de retorno funcionan conjuntamente. El tipo de retorno `int` de `getPrice` es una especie de promesa de que el cuerpo del método llevará a cabo algún tipo de acción que terminará por hacer que se calcule un valor entero y que se devuelva como resultado del método. En cierto modo, podemos pensar en que una llamada a un método es una especie de pregunta que se le hace a un objeto, y el valor de retorno proporcionado por el método es la respuesta que el objeto da a esa pregunta. En este caso, cuando se invoca el método `getPrice` en una máquina expendedora, la pregunta es: ¿cuánto cuestan los billetes? Una máquina expendedora no necesita realizar ningún cálculo para ser capaz de responder a esa pregunta, ya que tiene almacenada la respuesta en su campo `price`, por lo que el método responde simplemente devolviendo el valor de esa variable. A medida que vayamos desarrollando gradualmente clases más complejas, nos encontraremos inevitablemente con preguntas más complicadas que requerirán más trabajo para proporcionar la respuesta.

2.8 Métodos selectores y mutadores

A menudo, los métodos similares a los dos métodos "get" de `TicketMachine` (`getPrice` y `getBalance`) se denominan *métodos selectores* (o simplemente *selectores*). Esto se debe a que devuelven al llamante información acerca del estado de un objeto; proporcionan acceso a información acerca del estado del objeto. Un selector suele contener una instrucción de retorno, para poder devolver dicha información.

Concepto:

Los **métodos selectores** devuelven información acerca del estado de un objeto.

Existe confusión acerca de lo que realmente significa eso de "devolver un valor". Las personas tienden a crear que significa que el programa imprime alguna cosa, pero no es así en absoluto –veremos cómo se llevan a cabo las tareas de impresión cuando examinemos el método

`printTicket`. En lugar de ello, devolver un valor significa que se pasa una cierta información internamente entre dos partes diferentes del programa. Una parte del programa ha solicitado la información de un objeto mediante la invocación de un método y el valor de retorno es la forma que el objeto tiene de devolver dicha información al llamante.

Ejercicio 2.23 Compare la cabecera y el cuerpo del método `getBalance` con la cabecera y el cuerpo del método `getPrice`. ¿Qué diferencias hay entre ellos?

Ejercicio 2.24 Si una llamada a `getPrice` puede caracterizarse como "¿cuánto cuestan los billetes?", ¿cómo caracterizaría una llamada a `getBalance`?

Ejercicio 2.25 Si cambiamos el nombre de `getBalance` por `getAmount`, ¿será necesario cambiar también la instrucción de retorno en el cuerpo del método para que el código pueda compilarse? Pruébelo con BlueJ. ¿Qué nos dice esto acerca del nombre de un método selector y el nombre del campo asociado con él?

Ejercicio 2.26 Escriba un método selector `getTotal` en la clase `TicketMachine`. El nuevo método debe devolver el valor del campo `total`.

Ejercicio 2.27 Pruebe a eliminar la instrucción de retorno del cuerpo de `getPrice`. ¿Qué mensaje de error se obtiene al tratar de compilar la clase?

Ejercicio 2.28 Compare las cabeceras de los métodos `getPrice` y `printTicket` en el Código 2.1. Además de sus nombres, ¿cuál es la principal diferencia entre ellas?

Ejercicio 2.29 ¿Tienen instrucciones de retorno los métodos `insertMoney` y `printTicket`? ¿Por qué cree que puede ser esto? ¿Observa algo en sus cabeceras que pueda sugerir por qué no requieren instrucciones de retorno?

Concepto:

Los **métodos mutadores** cambian el estado de un objeto.

Los métodos `get` de una máquina expendedora realizan tareas similares: devolver el valor de uno de los campos del objeto correspondiente. Los métodos restantes –`insertMoney` y `printTicket`– desempeñan un papel mucho más significativo, principalmente porque *modifican* el valor de uno o más campos de un objeto máquina expendedora cada vez que se los invoca. A los métodos que modifican el estado de su objeto los denominamos *métodos mutadores* (o simplemente *mutadores*).

De la misma forma que podemos pensar en una llamada a un selector como si fuera una solicitud de información (una pregunta), podemos pensar en una llamada a un mutador como si fuera una solicitud para que un objeto cambie su estado. La forma más básica de mutador es aquella que admite un único parámetro cuyo valor se utiliza para sobrescribir directamente lo que haya almacenado en uno de los campos del objeto. Como complemento directo de los métodos "get", este conjunto de métodos se denominan a menudo métodos "set", aunque la clase `TicketMachine` no tiene ninguno de estos por el momento.

Un efecto distintivo de un mutador es que un objeto exhibirá a menudo un comportamiento ligeramente distinto antes y después de invocar a ese mutador. Podemos ilustrar este punto con el siguiente ejercicio.

Ejercicio 2.30 Cree una máquina expendedora con un precio de billete de su elección. Antes de hacer ninguna otra cosa, llame al método `getBalance`. A continuación, llame al método `insertMoney` (Código 2.6) y proporcione una cantidad de dinero positiva distinta de cero como parámetro real. Ahora vuelva a llamar a `getBalance`. Las dos llamadas a `getBalance` deberían mostrar salidas diferentes, porque la llamada a `insertMoney` ha tenido el efecto de cambiar el estado de la máquina, a través de su campo `balance`.

Código 2.6
El método `insertMoney`.

```
/**
 * Recibe una cierta cantidad de dinero en céntimos de un cliente.
 */
public void insertMoney(int amount)
{
    balance = balance + amount;
}
```

La cabecera de `insertMoney` tiene un tipo de retorno `void` y un único parámetro formal, `amount`, de tipo `int`. Un tipo de retorno `void` indica que el método no devuelve ningún valor al llamante. Este tipo de retorno es significativamente distinto a todos los demás tipos de retorno. En BlueJ, la diferencia más destacable es que no se muestra ningún cuadro de diálogo de valor de retorno después de una llamada a un método `void`. Dentro del cuerpo de un método `void`, esta diferencia se refleja en el hecho de que no hay instrucción de retorno.[2]

En el cuerpo de `insertMoney`, hay una única instrucción, que es otra forma de instrucción de asignación. Siempre analizaremos las instrucciones de asignación examinando primero el cálculo especificado en el lado derecho del símbolo de asignación. Aquí, su efecto consiste en calcular un valor que es igual a la suma del número contenido en el parámetro `amount` y del número contenido en el campo `balance`. Este valor combinado se asigna a continuación al campo `balance`. Por tanto, el efecto consiste en incrementar el valor de `balance` con el valor contenido en `amount`.[3]

Ejercicio 2.31 ¿Cómo podemos deducir, examinando simplemente la cabecera, que `setPrice` es un método y no un constructor?

```
public void setPrice(int cost)
```

---

[2] De hecho, Java permite que los métodos `void` contengan una forma especial de instrucción de retorno, en la que no hay ningún valor de retorno. Estas instrucciones tienen la forma

```
return;
```

y simplemente hacen que el método termine sin ejecutar ningún código ulterior.

[3] Sumar una cantidad al valor de una variable es una operación tan común, que existe un **operador de asignación compuesto** especial para hacer esto: +=. Por ejemplo:

```
balance += amount;
```

Ejercicio 2.32 Complete el cuerpo del método `setPrice` para que el método asigne el valor de su parámetro al campo `price`.

Ejercicio 2.33 Complete el cuerpo del siguiente método, cuyo propósito es sumar el valor de su parámetro a un campo denominado `score`.

```
/**
 * Incrementar la puntuación (score) con el número
 * de puntos indicado.
 */
public void increase(int points)
{
    ...
}
```

Ejercicio 2.34 ¿Es el método `increase` un mutador? En caso afirmativo, ¿cómo lo demostraría?

Ejercicio 2.35 Complete el siguiente método, cuyo propósito consiste en restar el valor de su parámetro de un campo denominado `price`.

```
/**
 * Reducir el precio en la cantidad (amount) indicada.
 */
public void discount(int amount)
{
    ...
}
```

2.9 Impresión desde métodos

El Código 2.7 muestra el método más complejo de la clase: `printTicket`. Para ayudarle a comprender las siguientes explicaciones, asegúrese de haber invocado este método con una máquina expendedora. Debería haber visto impreso en la ventana de terminal de BlueJ algo similar a lo siguiente:

```
##################
# The BlueJ Line
# Ticket
# 500 cents.
##################
```

Este es el método más largo que hemos visto hasta ahora, así que vamos a descomponerlo en partes más manejables:

- La cabecera indica que el método tiene un tipo de retorno `void` y que no requiere ningún parámetro.
- El cuerpo está compuesto por ocho instrucciones más los comentarios asociados.
- Las primeras seis instrucciones son responsables de imprimir lo que vemos en la ventana de terminal de BlueJ: cinco líneas de texto y una sexta línea en blanco.

Código 2.7
El método `printTicket`.

```
/**
 * Imprimir un billete y poner
 * el balance actual a cero.
 */
public void printTicket()
{
    // Simular la impresión de un billete.
    System.out.println("##################");
    System.out.println("# The BlueJ Line");
    System.out.println("# Ticket");
    System.out.println("# " + price + " cents.");
    System.out.println("##################");
    System.out.println();

    // Actualizar el total acumulado con el balance.
    total = total + balance;
    // Poner a cero el balance.
    balance = 0;
}
```

- La séptima instrucción suma el balance insertado por el cliente (mediante llamadas anteriores a `insertMoney`) al total de dinero acumulado por la máquina hasta el momento.
- La octava instrucción vuelve a poner a cero el balance mediante una instrucción de asignación básica dejando lista la máquina para el siguiente cliente.

Comparando la salida que aparece con las instrucciones que la han generado, es fácil ver que una instrucción como

```
System.out.println("# The BlueJ Line");
```

imprime literalmente la cadena de caracteres que aparece entre la pareja de caracteres de dobles comillas. La forma básica de una llamada a `println` es

`System.out.println(`*algo-que-queremos-imprimir*`);`

donde *algo-que-queremos-imprimir* puede sustituirse por cualquier cadena de caracteres arbitraria encerrada entre dobles comillas. Por ejemplo, no hay nada especial en el carácter "#" incluido en la cadena de caracteres –se trata simplemente de uno de los caracteres que queremos que se impriman.

Concepto:

El método **System.out.println** imprime su parámetro en el terminal de texto.

Todas las instrucciones de impresión del método `printTicket` son llamadas al método `println` del objeto `System.out` que está incorporado en el lenguaje Java, y lo que aparece entre los paréntesis es el parámetro de cada llamada al método, como cabría esperar. Sin embargo, en la cuarta instrucción, el parámetro real utilizado en `println` es algo más complicado y requiere algunas explicaciones adicionales.

```
System.out.println("# " + price + " cents.");
```

Lo que hace es imprimir el precio del billete con algunos caracteres adicionales a ambos lados de ese valor. Los dos operadores "+" se emplean para construir un único parámetro real, en forma de cadena de caracteres, a partir de tres componentes separados:

- el literal de cadena: "# " (fíjese en el carácter de espaciado después del símbolo de almohadilla);
- el valor del campo `price` (observe que no se usan comillas alrededor del nombre del campo, porque lo que queremos es el valor del campo, no su nombre);
- el literal de cadena: " cents." (observe el carácter de espaciado antes de la palabra "cents").

Cuando se utiliza entre una cadena y cualquier otra cosa, "+" es un operador de concatenación de cadenas (es decir, concatena o junta cadenas de caracteres con el fin de crear una nueva cadena) y no un operador de suma aritmética. Por tanto, el valor numérico de `price` se convierte en una cadena de caracteres y se concatena con las dos cadenas circundantes.

Observe que la llamada final a `println` no contiene ningún parámetro de cadena. Esto es algo perfectamente admisible y el resultado de llamar a ese método será dejar una línea en blanco entre la salida del método `printTicket` y cualquier otra cosa que se imprima posteriormente. Podrá ver fácilmente la línea en blanco si imprime un segundo billete.

Ejercicio 2.36 Escriba lo que imprimirá exactamente la siguiente instrucción:

```
System.out.println("My cat has green eyes.");
```

Ejercicio 2.37 Añada un método denominado `prompt` a la clase `TicketMachine`. Ese método debe tener un tipo de retorno `void` y ningún parámetro. El cuerpo del método debe imprimir la siguiente línea de salida:

```
Please insert the correct amount of money.
```

para informar al cliente de que introduzca la cantidad correcta de dinero.

Ejercicio 2.38 ¿Qué cree que se imprimiría si modificáramos la cuarta instrucción de `printTicket` de modo que `price` también se encierre entre comillas de la forma siguiente?

```
System.out.println("# " + "price" + " cents.");
```

Ejercicio 2.39 ¿Y que pasaría con la siguiente versión?

```
System.out.println("# price cents.");
```

Ejercicio 2.40 ¿Podríamos utilizar alguna de las dos versiones anteriores para mostrar el precio de los billetes en diferentes máquinas expendedoras? Explique su respuesta.

Ejercicio 2.41 Añada un método `showPrice` a la clase `TicketMachine`. Debe tener un tipo de retorno `void` y ningún parámetro. El cuerpo del método debe imprimir:

```
The price of a ticket is xyz cents.
```

para informar al cliente de cuál es el precio del billete. `xyz` debe sustituirse por el valor contenido en el campo `price` en el momento de invocar el método.

Ejercicio 2.42 Cree dos máquinas expendedoras con diferentes precios de billete. ¿Las llamadas a sus métodos `showPrice` muestran la misma salida, o diferente? ¿Cómo explica este efecto?

2.10 Resumen sobre los métodos

Resulta conveniente resumir en este punto unas cuantas características de los métodos, porque los métodos son fundamentales en los programas que vamos a escribir y a analizar en este libro. Los métodos implementan las acciones fundamentales realizadas por los objetos.

Un método con parámetros recibirá los datos que se le pasen desde la entidad que invoca a ese método y usará dichos datos para poder llevar a cabo una tarea concreta. Sin embargo, no todos los métodos utilizan parámetros; muchos hacen uso simplemente los datos almacenados en los campos del objeto para llevar a cabo su tarea.

Si un método tiene un tipo de retorno distinto de `void`, devolverá algún dato al lugar desde el que fue invocado –y dicho dato será utilizado, casi con total seguridad, en el llamante para realizar cálculos adicionales o para controlar la ejecución del programa. Muchos métodos, sin embargo, tienen un tipo de retorno `void` y no devuelven nada, aunque siguen realizando una tarea útil dentro del contexto de su objeto.

Los métodos selectores tienen tipos de retorno distintos de `void` y devuelven información acerca del estado de un objeto. Los métodos mutadores modifican el estado de un objeto. Los mutadores suelen tener parámetros, cuyos valores se utilizan en la modificación, aunque es perfectamente posible escribir un método mutador que no admita ningún parámetro.

2.11 Resumen de la máquina expendedora simple

Hemos examinado ya con un cierto grado de detalle la estructura interna de la clase que hemos empleado para modelar nuestra máquina expendedora simple. Hemos visto que la clase tiene una pequeña capa externa que proporciona un nombre a la clase y un cuerpo interno de mayor tamaño que contiene campos, un constructor y varios métodos. Los campos se utilizan para almacenar datos que permiten a los objetos mantener un estado que persiste entre llamadas sucesivas a los métodos. Los constructores se utilizan para configurar un estado inicial cuando se crea el objeto. Disponer de un estado inicial apropiado permitirá a los objetos responder adecuadamente a las llamadas a métodos que se produzcan inmediatamente después de la creación de esos objetos. Los métodos implementan el comportamiento definido para los objetos pertenecientes a esa clase. Los métodos selectores proporcionan información acerca del estado de un objeto y los métodos mutadores modifican el estado de un objeto.

Hemos visto que los constructores se distinguen de los métodos porque tienen el mismo nombre que la clase en la que están definidos. Los constructores como los métodos pueden aceptar parámetros, pero solo los métodos pueden tener un tipo de retorno. Los tipos de rctorno distintos dc `void` nos pcrmiten pasar un valor desde el interior de un método hacia el lugar desde el que el método fue invocado. Un método con un tipo de retorno distinto de `void` debe tener al menos una instrucción de retorno dentro de su cuerpo; a menudo, dicha instrucción será la última del método. Los constructores nunca tienen un tipo de retorno –ni siquiera `void`.

Antes de intentar realizar estos ejercicios, asegúrese de comprender bien cómo se comportan las máquinas expendedoras y cómo se implementa dicho comportamiento a través de los campos, el constructor y los métodos de la clase.

Ejercicio 2.43 Modifique el constructor de `TicketMachine` para que ya no admita un parámetro. En lugar de ello, el precio de los billetes debe estar fijado en 1000 céntimos. ¿Qué efecto tiene esto cuando se construyen objetos máquina expendedora en BlueJ?

Ejercicio 2.44 Defina dos constructores en la clase. Uno debe admitir un único parámetro que especifique el precio y el otro no debe admitir ningún parámetro y tiene que fijar para el precio un valor predeterminado que usted elija. Compruebe la implementación creando máquinas mediante los dos diferentes constructores.

Ejercicio 2.45 Implemente un método, `empty`, que simule el efecto de extraer todo el dinero de la máquina. Este método debe tener un tipo de retorno `void` y su cuerpo debe simplemente asignar el valor cero al campo `total`. ¿Necesita este método algún tipo de parámetro? Compruebe este método creando un máquina, insertando algo de dinero, imprimiendo algunos billetes, comprobando el total y luego vaciando la máquina. ¿Es el método `empty` un mutador o un selector?

2.12 Reflexiones sobre el diseño de la máquina expendedora

Gracias al estudio de los detalles internos de la clase `TicketMachine`, habrá podido apreciar lo inadecuada que sería esta máquina expendedora en el mundo real. Tiene múltiples deficiencias:

- No efectúa ninguna comprobación de que el cliente ha introducido el suficiente dinero para pagar el billete.
- No devuelve dinero si el cliente introduce más dinero del que cuesta el billete.
- No efectúa ninguna comprobación para ver si el cliente está introduciendo cantidades de dinero lógicas. Compruebe lo que sucede si introduce, por ejemplo, un valor negativo.
- No comprueba que el precio del billete pasado a su constructor sea lógico.

Si pudiéramos remediar estos problemas, tendríamos un software bastante más funcional que podría servir como base para el control de una máquina expendedora del mundo real.

En las siguientes secciones, vamos a examinar la implementación de una clase mejorada de máquina expendedora, que tratará de resolver algunos de los defectos de la implementación simple. Abra el proyecto *better-ticket-machine*. Como antes, este proyecto contiene una única clase: `TicketMachine`. Antes de ver los detalles internos de esta clase, experimente con ella creando algunas instancias y viendo si se aprecian diferencias de comportamiento entre esta versión y la versión anterior más sencilla.

Una diferencia específica es que la nueva versión tiene un método adicional, `refundBalance`. Observe lo que sucede cuando se invoca ese método.

Código 2.8
Una clase TicketMachine más sofisticada.

```
/**
 * TicketMachine modela una máquina expendedora que emite billetes
 * de un único precio.
 * El precio de un billete se especifica mediante el constructor.
 * Las instancias comprobarán que el usuario introduzca solo
 * cantidades lógicas de dinero y solo imprimirán un billete
 * si se ha introducido el dinero suficiente.
 * @author David J. Barnes y Michael Kölling
 * @version 2011.07.31
 */
public class TicketMachine
{
    // Precio de un billete en esta máquina.
    private int price;
    // Cantidad de dinero insertada hasta el momento por el cliente.
    private int balance;
    // Cantidad total de dinero acumulada por esta máquina
    private int total;

    /**
     * Crea una máquina que emite billetes de un determinado precio.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * Devuelve el precio de un billete.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Devuelve la cantidad de dinero ya insertada para el
     * siguiente billete.
     */
    public int getBalance()
    {
        return balance;
    }

    /**
     * Recibe una cantidad de dinero en céntimos de un cliente.
     * Comprueba que la cantidad introducida sea lógica.
     */
```

Código 2.8 (continuación)
Una clase TicketMachine más sofisticada.

```
    public void insertMoney(int amount)
    {
        if(amount > 0) {
            balance = balance + amount;
        }
        else {
            System.out.println("Use a positive amount rather than: " +
                               amount);
        }
    }

    /**
     * Imprime un billete si se ha insertado el dinero suficiente
     * y resta el precio del billete del balance actual. Imprime
     * un mensaje de error si hace falta más dinero.
     */
    public void printTicket()
    {
        if(balance >= price) {
            // Simula la impresión de un billete.
            System.out.println("##################");
            System.out.println("# The BlueJ Line");
            System.out.println("# Ticket");
            System.out.println("# " + price + " cents.");
            System.out.println("##################");
            System.out.println();

            // Actualiza el total acumulado con el precio.
            total = total + price;
            // Resta el precio del balance.
            balance = balance - price;
        }
        else {
            System.out.println("You must insert at least: " +
                               (price - balance) + " cents.");
        }
    }

    /**
     * Devolver el dinero del balance.
     * Poner a cero el balance.
     */
    public int refundBalance()
    {
        int amountToRefund;
        amountToRefund = balance;
        balance = 0;
        return amountToRefund;
    }
}
```

2.13 Tomas de decisión: la instrucción condicional

El Código 2.8 muestra los detalles internos de la definición mejorada de la clase `TicketMachine`. Gran parte de esta definición ya le resultará familiar a partir de las explicaciones dadas para la máquina expendedora simple. Por ejemplo, el envoltorio externo que da nombre a la clase es el mismo, porque hemos elegido dar a esta clase el mismo nombre. Además, contiene los mismos tres campos para mantener el estado del objeto y esos tres campos se han declarado de la misma forma. El constructor y los dos métodos `get` también son iguales que antes.

El primer cambio significativo puede verse en el método `insertMoney`. Habíamos visto que el principal problema con la máquina expendedora simple era que no comprobaba ciertas condiciones. Una de esas comprobaciones que faltaba era la relativa a la cantidad de dinero insertada por un cliente, ya que era posible introducir una cantidad negativa de dinero. Hemos puesto remedio a esta situación haciendo uso de una *instrucción condicional*, para comprobar que la cantidad insertada tiene un valor mayor que cero.

```
if(amount > 0) {
    balance = balance + amount;
}
else {
    System.out.println("Use a positive amount rather than: " +
                       amount);
}
```

Concepto:

Una **instrucción condicional** lleva a cabo una de dos posibles acciones basándose en el resultado de una prueba.

Las instrucciones condicionales también se conocen con el nombre de *instrucciones if*, debido a la palabra clave usada en la mayoría de los lenguajes de programación para implementarlas. Una instrucción condicional nos permite llevar a cabo una de dos posibles acciones basándonos en el resultado de una prueba o comprobación. Si la comprobación es verdadera entonces hacemos una cosa; en caso contrario, hacemos algo distinto. Este tipo de decisión entre dos alternativas nos resulta familiar ya que son similares a esas decisiones tan frecuentes que tomamos en la vida cotidiana: por ejemplo, si me queda suficiente dinero, saldré a cenar fuera; en caso contrario, me quedaré en casa y veré una película. Una instrucción condicional tiene la forma general descrita en el siguiente *pseudocódigo*:

`if(`*realizar una prueba que dé un resultado verdadero o falso*`) {`
    Ejecutar estas instrucciones si la prueba dio un resultado verdadero
`}`
`else {`
    Ejecutar estas instrucciones si la prueba dio un resultado falso
`}`

Ciertas partes de este pseudocódigo son componentes reales de Java y esos componentes aparecen en casi todas las instrucciones condicionales –nos referimos en concreto a las palabras clave `if` y `else`, a los paréntesis que encierran la comprobación que hay que utilizar y a las llaves que delimitan los dos bloques de instrucciones–, mientras que las otras tres partes que se muestran en cursiva tendrán una implementación diferente en cada situación concreta que se desee programar.

Es importante observar que, después de evaluar la comprobación condicional, solo se ejecuta uno de los dos bloques de instrucciones situados después de esa comprobación. Por tanto, en el ejemplo del método `insertMoney`, después de comprobar la cantidad de dinero introducido lo que haremos será, o bien sumar esa cantidad al balance o bien imprimir el mensaje de error. La comprobación utiliza el *operador mayor que*, ">", para comparar el valor de `amount` con cero. Si el valor es mayor que cero, se suma al balance. Si no es mayor que cero, entonces se imprime un mensaje de error. Utilizando una instrucción condicional, lo que hacemos en la práctica es proteger las modificaciones de `balance` en aquellos casos en los que el parámetro no representa una cantidad válida. En el Apéndice C puede encontrar más detalles acerca de otros operadores Java. Los más obvios que podemos mencionar aquí son: "<" (menor que), "<=" (menor o igual que) y ">=" (mayor o igual que). Todos ellos se emplean para comparar dos valores numéricos, como en el método `printTicket`.

Concepto:

Las **expresiones booleanas** solo tienen dos posibles valores: verdadero (true) y falso (false). Se las utiliza a menudo a la hora de controlar la elección entre las dos rutas de ejecución especificadas en una instrucción condicional.

La comprobación utilizada en una instrucción condicional es un ejemplo de *expresión booleana.* Anteriormente en el capítulo, hemos presentado expresiones aritméticas que generaban resultados numéricos. Una expresión booleana sólo puede tomar dos posibles valores (`true` o `false`): el valor de `amount` solo puede, o ser mayor que cero (`true`) o no ser mayor que cero (`false`). Una instrucción condicional hace uso de esos dos posibles valores para elegir entre dos acciones distintas.

Ejercicio 2.46 Compruebe que el comportamiento que hemos explicado aquí es apropiado, creando una instancia de `TicketMachine` e invocando `insertMoney` con diversos parámetros reales. Compruebe el balance tanto antes como después de llamar a `insertMoney`. ¿Cambia el balance alguna vez en aquellos casos en los que se imprime un mensaje de error? Trate de predecir lo que sucedería si introdujera el valor cero como parámetro y compruebe después si su predicción es correcta.

Ejercicio 2.47 Prediga lo que cree que sucedería si modificamos la comprobación en `insertMoney` para que utilice el *operador mayor o igual que:*

```
if(amount >= 0)
```

Compruebe su predicción ejecutando algunas pruebas. ¿Qué diferencia introduce esto en el comportamiento del método?

Ejercicio 2.48 Escriba de nuevo la instrucción if-else para que se imprima el mensaje de error si la expresión booleana es verdadera y el balance se incremente si la expresión es falsa. Obviamente, tendrá que escribir de nuevo la condición para hacer que el método funcione en esta situación en la que hemos intercambiado los dos bloques de instrucciones.

Ejercicio 2.49 En el proyecto *figures* que hemos visto en el Capítulo 1 utilizamos un campo de tipo `boolean` para controlar una cierta característica de los objetos círculo. ¿Cuál era esa característica? ¿Se trata de una característica adecuada para ser controlada por un tipo de dato que solo dispone de dos valores distintos?

2.14 Ejemplo adicional de instrucción condicional

El método printTicket contiene un ejemplo más de instrucción condicional. He aquí su aspecto resumido:

```
if(balance >= price) {
    Se omiten los detalles de impresión.
    // Actualizar el total recaudado con el precio.
    total = total + price;
    // Restar el precio del balance.
    balance = balance - price;
}
else {
    System.out.println("You must insert at least: " +
                       (price - balance) + " more cents.");
}
```

Queremos remediar el hecho de que la versión simple de la máquina expendedora no hace ninguna comprobación de que un cliente ha introducido dinero suficiente como para entregarle un billete. En esta nueva versión se comprueba que el valor del campo balance es al menos igual al valor del campo price. Si lo es, entonces será correcto imprimir un billete. Si no lo es, lo que hacemos es imprimir en su lugar un mensaje de error.

La impresión del mensaje de error sigue exactamente el mismo patrón que ya hemos visto para la impresión de los billetes en el método printTicket; simplemente, la instrucción correspondiente es algo más larga.

```
System.out.println("You must insert at least: " +
                   (price - balance) + " more cents.");
```

El único parámetro real del método println está formado por una concatenación de tres elementos: dos literales de cadena antes y después de un valor numérico. En este caso, el valor numérico es una resta que se ha encerrado entre paréntesis para indicar que lo que queremos concatenar con las dos cadenas es el valor resultante de esa resta.

Ejercicio 2.50 En esta versión de printTicket, también hacemos algo ligeramente distinto con los campos total y balance. Compare la implementación del método en el Código 2.1 con la del Código 2.8, para ver si puede detectar esas diferencias. A continuación, compruebe que ha entendido bien lo que sucede, experimentado con BlueJ.

Ejercicio 2.51 ¿Es posible eliminar la parte else de la instrucción if en el método printTicket (es decir, eliminar la palabra else y el bloque asociado a la misma)? Trate de hacer esto y compruebe si el código sigue compilándose. ¿Qué sucede ahora si intenta imprimir un billete sin insertar ningún dinero?

El método printTicket reduce el valor de balance restando de él el valor de price. Como consecuencia, si un cliente introduce más dinero que el precio del billete, entonces quedará

algo de dinero en el balance y se podrá utilizar para completar el precio de un segundo billete. Alternativamente, el cliente puede pedir que se le devuelva el saldo restante, y eso es lo que hace el método `refundBalance`, como veremos en la siguiente sección.

2.15 Representación visual del ámbito

Ya habrá observado que BlueJ muestra el código fuente con algunos detalles de formato adicionales: concretamente, sitúa recuadros coloreados alrededor de algunos elementos; esos recuadros no están reproducidos en los ejemplos de código que se muestran en el libro (Figura 2.5).

Figura 2.5
Representación visual del ámbito en el editor de BlueJ.

```
TicketMachine
Compile  Undo  Cut  Copy  Paste  Find...  Close        Source Code

        }
    }

    /**
     * Print a ticket if enough money has been inserted, and
     * reduce the current balance by the ticket price. Print
     * an error message if more money is required.
     */
    public void printTicket()
    {
        if(balance >= price) {
            // Simulate the printing of a ticket.
            System.out.println("##################");
            System.out.println("# The BlueJ Line");
            System.out.println("# Ticket");
            System.out.println("# " + price + " cents.");
            System.out.println("##################");
            System.out.println();

            // Update the total collected with the price.
            total = total + price;
            // Reduce the balance by the prince.
            balance = balance - price;
        }
        else {
            System.out.println("You must insert at least: " +
                               (price - balance) + " more cents.");

        }
    }

    /**
     * Return the money in the balance.
     * The balance is cleared.

saved
```

Estas indicaciones de color se conocen con el nombre de *representación visual del ámbito* y pueden ayudarnos a clarificar las unidades lógicas del programa. Un *ámbito* (también denominado *bloque*) es una unidad de código que normalmente está encerrada entre llaves. El cuerpo completo de una clase es un ámbito, como también lo son el cuerpo de cada método y las partes *if* y *else* de una instrucción condicional.

Como puede ver, los ámbitos están a menudo anidados: la instrucción if se encuentra dentro de un método, que a su vez se encuentra dentro de una clase. BlueJ ayuda a diferenciar los distintos ámbitos empleando distintos colores.

Uno de los errores más comunes en el código generado por los programadores principiantes es no emparejar adecuadamente las llaves que definen los distintos bloques –bien porque las colocan en lugares inadecuados o bien porque se olvidan de una de las dos llaves. Hay dos cosas que ayudan enormemente a evitar este tipo de error:

- Preste atención a la hora de utilizar correctamente el sangrado en su código. Cada vez que comience un nuevo ámbito (después de una llave de apertura), aumente un nivel más

el sangrado del código situado a continuación. Cerrar el ámbito devolverá el sangrado a su nivel anterior. Si el sangrado estuviera completamente liado, utilice la función "Auto-layout" de BlueJ (puede encontrarla en el menú del editor) para corregir la situación.

- Preste atención a la representación visual de los distintos ámbitos. Rápidamente se acostumbrará al aspecto que tiene que tener un código bien estructurado. Trate de eliminar una llave en el editor o de añadir otra en una posición arbitraria y observe cómo cambian los colores. Acostúmbrese a reconocer de manera rápida cuándo los ámbitos no tienen el aspecto adecuado.

Ejercicio 2.52 Después de imprimir un billete, ¿podría llegar a ser negativo el valor contenido en el campo `balance`, al restarle el valor contenido en `price`? Justifique su respuesta.

Ejercicio 2.53 Hasta ahora, hemos presentado dos operadores aritméticos, + y -, que pueden utilizarse como parte de las **expresiones aritméticas** en Java. Consulte el Apéndice C para ver qué otros operadores hay disponibles.

Ejercicio 2.54 Escriba una instrucción de asignación que almacene el resultado de multiplicar dos variables, `price` y `discount`, en una tercera variable, `saving`. Con esa instrucción podemos, por ejemplo, calcular el ahorro que se obtiene al aplicar un descuento a un cierto precio.

Ejercicio 2.55 Escriba una instrucción de asignación que divida el valor de `total` entre el valor de `count` y almacene el resultado en `mean`. Con esa instrucción podemos, por ejemplo, calcular la media del precio, dividiendo el total entre el número de billetes emitidos.

Ejercicio 2.56 Escriba una instrucción condicional que compare el valor de `price` con el valor de `budget`. Si `price` es mayor que `budget`, entonces imprima el mensaje "Too expensive" para indicar que el precio es excesivo; en caso contrario, imprima el mensaje "Just right", indicativo de que tenemos el suficiente dinero para pagar el precio indicado.

Ejercicio 2.57 Modifique su respuesta al ejercicio anterior para que el mensaje incluya el valor de `budget` (es decir, del presupuesto disponible) si el precio es demasiado alto.

2.16 Variables locales

Hasta ahora, nos hemos encontrado con dos tipos diferentes de variables: campos (variables de instancia) y parámetros. Ahora vamos a introducir un tercer tipo. Lo que tienen en común todos estos tipos de variable es que almacenan datos, pero cada tipo de variable desempeña un papel diferente.

En la Sección 2.6 hemos dicho que el cuerpo de un método (o, en general, cualquier *bloque*) puede contener tanto declaraciones como instrucciones. Sin embargo, hasta ahora, ninguno de los métodos que hemos examinado contenía ninguna declaración. El método `refundBalance` (reembolsar saldo) contienen tres instrucciones y una única declaración. La declaración introduce un nuevo tipo de variable:

```
public int refundBalance()
{
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

Concepto:

Una **variable local** es una variable declarada y utilizada dentro un único método. Su ámbito y su tiempo de vida están limitados a los del propio método.

¿Qué tipo de variable es `amountToRefund`, que indica la cantidad de dinero que hay que devolver al cliente? Sabemos que no es un campo, porque los campos se definen fuera de los métodos. Tampoco es un parámetro, ya que los parámetros siempre se definen en la cabecera del método. La variable `amountToRefund` es lo que conocemos como *variable local,* porque se define *dentro* del cuerpo de un método.

Las declaraciones de variables locales parecen similares a las declaraciones de campos, pero las palabras clave `private` y `public` nunca aparecen en la declaración. Los constructores también pueden tener variables locales. Al igual que los parámetros formales, las variables locales tienen un ámbito que está limitado a las instrucciones del método al que pertenecen. Su tiempo de vida coincide con el tiempo durante el cual se está ejecutando el método: se crean cuando se invoca un método y se destruyen cuando el método termina.

Puede que se esté preguntando para qué hacen faltan las variables locales si ya disponemos de campos. Las variables locales se usan principalmente como almacenamiento temporal, para ayudar a un método a completar su tarea; podemos considerarlas como un almacenamiento de datos para un único método. Por el contrario, los campos se utilizan para almacenar datos que permanecen durante toda la vida de un objeto completo. Los datos almacenados en campos son accesibles para todos los métodos del objeto. Tenemos que intentar evitar declarar como campos aquellas variables que solo tienen realmente un uso local (en el nivel de método), es decir, variables cuyos valores no necesitan recordarse más allá de una única llamada al método. Por tanto, incluso aunque dos o más métodos de una misma clase utilicen variables locales con un propósito similar, no sería apropiado definirlas como campos si sus valores no necesitan persistir más allá del momento en que termina la ejecución de esos métodos.

En el método `refundBalance`, la variable `amountToRefund` se utiliza brevemente para almacenar el valor de `balance` inmediatamente antes de ponerlo a cero. El método devuelve entonces ese valor del balance que ha almacenado en la variable. Los siguientes ejercicios nos ayudarán a ilustrar por qué hace falta aquí una variable local; para ello, intentaremos escribir el método `refundBalance` sin usar dicha variable.

Ejercicio 2.58 ¿Por qué la siguiente versión de `refundBalance` no da los mismos resultados que la original?

```
public int refundBalance()
{
    balance = 0;
    return balance;
}
```

¿Qué pruebas podría hacer para demostrar que no proporciona los mismos resultados?

Ejercicio 2.59 ¿Qué sucede si trata de compilar la clase `TicketMachine` con la siguiente versión de `refundBalance`?

```
public int refundBalance()
{
    return balance;
    balance = 0;
}
```

¿Qué característica de las instrucciones de retorno ayuda a explicar por qué esta versión no puede compilarse?

Ejercicio 2.60 ¿Qué tiene de incorrecto la siguiente versión del constructor de `TicketMachine`?

```
public TicketMachine(int cost)
{
    int price = cost;
    balance = 0;
    total = 0;
}
```

Pruebe esta versión en el proyecto *better-ticket-machine*. ¿Se compila correctamente esta versión? Cree un objeto e inspeccione después sus campos. ¿Observa algo incorrecto en el inspector al examinar el valor del campo `price` en esta versión? ¿Podría explicar por qué sucede esto?

Es bastante común inicializar las variables locales en el momento de declararlas. Por ello, podríamos abreviar las dos primeras instrucciones de `refundBalance` y escribir simplemente

```
int amountToRefund = balance;
```

pero sigue siendo importante recordar que aquí se están ejecutando dos pasos distintos: declarar la variable `amountToRefund` y proporcionarle un valor inicial.

Error común Una variable local del mismo nombre que un campo impedirá que se pueda acceder a ese campo desde dentro de constructor o método. Consulte la Sección 3.12.2 para ver cómo solucionar esto en caso necesario.

2.17 Campos, parámetros y variables locales

Con la introducción de `amountToRefund` en el método `refundBalance`, hemos visto ya tres tipos distintos de variables: campos, parámetros formales y variables locales. Es importante comprender las similitudes y diferencias entre estos tres tipos. He aquí un resumen de sus características:

- Los tres tipos de variables son capaces de almacenar un valor que se corresponda con su tipo definido. Por ejemplo, una variable cuyo tipo definido sea `int` permite que esa variable almacene un valor entero.
- Los campos se definen fuera de los constructores y métodos.

- Los campos se utilizan para almacenar datos que persisten durante toda la vida de un objeto. Por ello, mantienen el estado actual de un objeto. Tienen un tiempo de vida que coincide con la duración del objeto al que pertenecen.
- Los campos tienen un ámbito que coincide con la clase: son accesibles desde cualquier punto de la clase a la que pertenecen, de modo que se pueden utilizar dentro de cualquiera de los constructores o métodos de la clase en la que han sido definidos.
- Mientras se definan como privados (`private`) no se podrá acceder a los campos desde ningún punto situado fuera de la clase en la que están definidos.
- Los parámetros formales y las variables locales solo persisten mientras que se está ejecutando un constructor o método. Su tiempo de vida coincide con la duración de una única invocación, por lo que sus valores se pierden entre invocaciones sucesivas. Desde ese punto de vista, actúan como ubicaciones de almacenamiento temporal, no permanente.
- Los parámetros formales se definen en la cabecera de un constructor o método. Reciben sus valores del exterior, siendo inicializados de acuerdo con los valores de los parámetros reales que forman parte de la llamada al constructor o al método.
- Los parámetros formales tienen un ámbito que está limitado al constructor o método en los que se los define.
- Las variables locales se definen dentro del cuerpo de un constructor o método. Solo pueden inicializarse y utilizarse dentro del cuerpo del constructor o método en el que se las define. Las variables locales deben inicializarse antes de poder ser utilizadas en una expresión –no se las proporciona un valor predeterminado.
- Las variables locales tienen un ámbito que está limitado al bloque en el que están definidas. No se puede acceder a ellas desde ningún punto situado fuera de dicho bloque.

Ejercicio 2.61 Añada un nuevo método, `emptyMachine`, diseñado para simular el vaciado del dinero acumulado en la máquina. Ese método debe reinicializar el campo `total` a cero, pero también devolver el valor que estuviera almacenado en `total` antes de la reinicialización.

Ejercicio 2.62 Escriba de nuevo el método `printTicket` para que declare una variable local, `amountLeftToPay`, que debe indicar el dinero que falta para completar el precio del billete. Dicha variable deberá inicializarse para que contenga la diferencia entre `price` y `balance`. Reescriba la comprobación en la instrucción condicional para que se compruebe el valor de `amountLeftToPay`. Si su valor es menor o igual que cero, deberá imprimirse un billete; en caso contrario, hay que imprimir un mensaje de error que especifique la cantidad de dinero que aún falta por introducir. Compruebe que el código que haya escrito para verificar que se comporta exactamente de la misma manera que la versión original. Asegúrese de llamar al método más de una vez, cuando la máquina se encuentre en diferentes estados, con el fin de que las dos partes de la instrucción condicional se ejecuten en diferentes ocasiones.

Ejercicio 2.63 *Ejercicio avanzado.* Suponga que queremos que un único objeto `TicketMachine` sea capaz de emitir billetes de precios diferentes. Por ejemplo, los usuarios pueden pulsar un botón en la máquina física para seleccionar un precio con descuento. ¿Qué métodos y/o campos adicionales tendríamos que incluir en `TicketMachine` para permitir este tipo de funcionalidad? ¿Cree que habría que cambiar también muchos de los métodos existentes?

Guarde el proyecto *better-ticket-machine* con un nuevo nombre e implemente sus cambios en el nuevo proyecto.

2.18 Resumen de la máquina expendedora mejorada

Al desarrollar una versión mejorada de la clase `TicketMachine`, hemos sido capaces de resolver los principales defectos de la versión simple. Al hacerlo, hemos presentado dos nuevas estructuras del lenguaje: la instrucción condicional y las variables locales.

- Una instrucción condicional nos proporciona un medio de realizar una comprobación para luego, dependiendo del resultado de esa comprobación, ejecutar una de dos posibles acciones distintas.
- Las variables locales nos permiten calcular y almacenar valores de forma temporal dentro de un constructor o un método. Contribuyen al comportamiento implementado por el método en el que se las define, pero sus valores se pierden después de que finaliza la ejecución de ese constructor o método.

Puede encontrar más detalles sobre la instrucciones condicionales y sobre la forma que pueden adoptar sus comprobaciones en el Apéndice D.

2.19 Ejercicios de autoevaluación

En este capítulo hemos cubierto muchas nuevas materias y hemos presentado muchos conceptos nuevos. Continuaremos profundizando sobre estos temas en los siguientes capítulos, así que es importante estar seguros de que nuestra comprensión es la adecuada. Trate de resolver los siguientes ejercicios con lápiz y papel, como forma de comprobar que está habituado a la terminología presentada en este capítulo. No se disuada de hacerlos por el hecho de que le sugiramos que los haga en papel y no en BlueJ. Es conveniente practicar probando ciertas cosas sin un compilador.

Ejercicio 2.64 Indique el nombre y el tipo de retorno de este método:

```
public String getCode()
{
    return code;
}
```

Ejercicio 2.65 Indique el nombre de este método y el nombre y el tipo de su parámetro:

```
public void setCredits(int creditValue)
{
    credits = creditValue;
}
```

Ejercicio 2.66 Escriba el envoltorio externo de una clase denominada `Person`. Recuerde incluir las llaves que marcan el principio y el fin del cuerpo de la clase, pero por lo demás deje dicho cuerpo vacío.

Ejercicio 2.67 Escriba las definiciones para los siguientes campos:

- un campo denominado `name` de tipo `String`
- un campo de tipo `int` denominado `age`

- un campo de tipo `String` denominado `code`
- un campo denominado `credits` de tipo `int`

Ejercicio 2.68 Escriba un constructor para una clase denominada `Module`. El constructor debe admitir un único parámetro de tipo `String` denominado `moduleCode`. El cuerpo del constructor debe asignar el valor de su parámetro a un campo denominado `code`. No es necesario incluir la definición de `code`, simplemente el texto del constructor.

Ejercicio 2.69 Escriba un constructor para una clase denominada `Person`. El constructor debe admitir dos parámetros. El primero de tipo `String` y nombre `myName`. Y el segundo de tipo `int` y nombre `myAge`. El primer parámetro debe emplearse para definir el valor de un campo denominado `name`, mientras que el segundo debe configurar un campo de nombre `age`. No tiene que incluir las definiciones de los campos, sino simplemente el texto del constructor.

Ejercicio 2.70 Corrija el error en este método:

```
public void getAge()
{
    return age;
}
```

Ejercicio 2.71 Escriba un método selector denominado `getName` que devuelva el valor de un campo de nombre `name`, cuyo tipo es `String`.

Ejercicio 2.72 Escriba un método mutador denominado `setAge` que acepte un único parámetro de tipo `int` y configure el valor de un campo denominado `age`.

Ejercicio 2.73 Escriba un método denominado `printDetails` para una clase que tenga un campo de tipo `String` denominado `name`. El método `printDetails` debe imprimir una cadena de la forma "The name of this person is" (el nombre de esta persona es), seguida del valor del campo `name`. Por ejemplo, si el valor del campo `name` es "Helen", entonces `printDetails` imprimirá:

```
The name of this person is Helen
```

Si ha conseguido completar la mayor parte de estos ejercicios o todos ellos, entonces puede tratar de crear un nuevo proyecto en BlueJ y realizar su propia definición de clase para modelar a una persona; llamaremos a esa clase `Person`. La clase podría tener campos para almacenar el nombre y la edad de una persona, por ejemplo. Si no está seguro de cómo completar alguno de los ejercicios anteriores, repase las secciones previas del capítulo y el código fuente de `TicketMachine`, para revisar los conceptos sobre los que tenga dudas. En la sección siguiente proporcionamos material de repaso adicional.

2.20 Revisión de un ejemplo familiar

Llegados a este punto del capítulo, nos hemos topado con una gran cantidad de nuevos conceptos. Como refuerzo para comprenderlos, vamos a volver a examinar algunos de ellos en un contexto distinto pero familiar. Sin embargo, mientras hacemos esto, esté atento a la aparición de un par de nuevos conceptos que posteriormente analizaremos con mayor detalle en otros capítulos.

Abra el proyecto *lab-classes* que presentamos en el Capítulo 1 y examine la clase `Student` en el editor (Código 2.9).

Código 2.9
La clase `Student`.

```
/**
 * La clase Student representa a un estudiante en un sistema
 * de administración de alumnos.
 * Almacena los detalles de los estudiantes que son relevantes
 * en nuestro contexto.
 *
 * @author Michael Kölling y David Barnes
 * @version 2011.07.31
 */
public class Student
{
    // Nombre completo del estudiante.
    private String name;
    // ID del estudiante.
    private String id;
    // Créditos que el estudiante ha cubierto hasta ahora.
    private int credits;

    /**
     * Crear un nuevo estudiante con un nombre y un número
     * de ID dados.
     */
    public Student(String fullName, String studentID)
    {
        name = fullName;
        id = studentID;
        credits = 0;
    }

    /**
     * Devuelve el nombre completo de este estudiante.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Establece un nuevo nombre para este estudiante.
     */
    public void changeName(String newName)
    {
        name = newName;
    }
```

Código 2.9
(*continuación*)
La clase `Student`.

```
    /**
     * Devuelve el ID de este estudiante.
     */
    public String getStudentID()
    {
        return id;
    }
    /**
     * Añadir créditos a los créditos acumulados del estudiante.
     */
    public void addCredits(int newCreditPoints)
    {
        credits += newCreditPoints;
    }

    /**
     * Devolver el número de créditos que este estudiante
     * ha acumulado.
     */
    public int getCredits()
    {
        return credits;
    }

    /**
     * Devolver el nombre de inicio de sesión de este estudiante.
     * El nombre de inicio de sesión es una combinación
     * de los cuatro primeros caracteres del nombre del
     * estudiante y los tres primeros caracteres
     * de su número de ID.
     */
    public String getLoginName()
    {
        return name.substring(0,4) +
               id.substring(0,3);
    }
    /**
     * Imprimir el nombre y el número de ID del estudiante
     * en el terminal de salida.
     */
    public void print()
    {
        System.out.println(name + ", student ID: " + id +
                           ", credits: " + credits);
    }
}
```

En este pequeño ejemplo, los elementos de información que queremos almacenar para un estudiante son su nombre, su identificador (ID) y el número de créditos que ha obtenido hasta el momento. Toda esta información es persistente durante su vida como estudiante, incluso aunque parte de ella vaya cambiando durante ese tiempo (el número de créditos). Queremos almacenar, por tanto, esta información en campos para representar el estado de cada estudiante.

La clase contiene tres campos: `name`, `id` y `credits`. Cada uno de estos se inicializa en el único constructor existente. Los valores iniciales de los dos primeros se establecen a partir de los parámetros pasados al constructor. Cada uno de los campos tiene un método selector `get` asociado, pero solo `name` y `credits` tienen métodos mutadores asociados. Esto quiere decir que el valor del campo `id` permanece fijo una vez que el objeto ha sido construido. Si el valor de un campo no puede modificarse después de haberlo inicializado, decimos que es *inmutable*. En ocasiones, hacemos inmutable el estado completo de un objeto después de construirlo; la clase `String` es un ejemplo importante de este tipo de caso.

2.21 Invocación de métodos

El método `getLoginName` ilustra una nueva característica que merece la pena analizar:

```
public String getLoginName()
{
    return name.substring(0,4) +
           id.substring(0,3);
}
```

Aquí podemos ver dos cosas distintas:

- Se están invocando métodos de otros objetos, siendo el caso de que los métodos devuelven un resultado.
- Se utiliza el resultado devuelto como parte de una expresión.

Tanto `name` como `id` son objetos de tipo `String` y la clase `String` tiene un método, `substring`, que tiene la siguiente cabecera:

```
/**
 * Devuelve una nueva cadena que contiene los caracteres de esta
 * cadena comprendidos entre beginIndex y (endIndex-1).
 */
public String substring(int beginIndex, int endIndex)
```

Un valor de índice de cero representa el primer carácter de una cadena, por lo que `getLoginName` toma los cuatro primeros caracteres de la cadena `name` y los tres primeros caracteres de la cadena `id` y luego los concatena para formar una nueva cadena de caracteres. Esta nueva cadena se devuelve como resultado del método. Por ejemplo, si `name` es la cadena `"Leonardo da Vinci"` e `id` es la cadena `"468366"`, entonces este método devolvería la cadena `"Leon468"`.

Aprenderemos más acerca de las llamadas a métodos entre objetos en el Capítulo 3.

Ejercicio 2.74 Dibuje una imagen de la forma mostrada en la Figura 2.3, que represente el estado inicial de un objeto `Student` después de su construcción, con los siguientes valores de parámetros reales:

```
new Student("Benjamin Jonson", "738321")
```

Ejercicio 2.75 ¿Qué devolvería `getLoginName` para un estudiante cuyo nombre (`name`) fuera `"Henry Moore"` y cuyo `id` fuera `"557214"`?

Ejercicio 2.76 Cree un objeto `Student` con nombre `"djb"` e `id` `"859012"`. ¿Qué sucede cuando se llama a `getLoginName` con este estudiante? ¿Por qué cree que ocurre esto?

Ejercicio 2.77 La clase `String` define un método selector `length` con la siguiente cabecera:

```
/**
 * Devuelve el número de caracteres que forman esta cadena.
 */
public int length()
```

de manera que un ejemplo de su uso con la variable de tipo `String` y nombre `fullName` sería:

```
fullName.length()
```

Añada instrucciones condicionales al constructor de `Student` para imprimir un mensaje de error si la longitud del parámetro `fullName` es menor de cuatro caracteres o si la longitud del parámetro `studentId` es menor de tres caracteres. Sin embargo, el constructor debe de todos modos utilizar esos parámetros para configurar los campos `name` e `id`, incluso aunque se imprima el mensaje de error. *Sugerencia:* utilice instrucciones if de la siguiente forma (es decir, sin parte `else`) para imprimir los mensajes de error.

```
if(realizar una comprobación con uno de los parámetros) {
    Imprimir un mensaje de error si el resultado de
    la comprobación es verdadero
}
```

Si fuera necesario, consulte el Apéndice D para ver detalles adicionales acerca de los diferentes tipos de instrucciones condicionales.

Ejercicio 2.78 *Ejercicio avanzado.* Modifique el método `getLoginName` de `Student` de manera que siempre genere un nombre de inicio de sesión, incluso si el campo `name` o el campo `id` no tienen la longitud adecuada. Para cadenas de longitud inferior a la requerida, utilice la cadena completa.

2.22 Experimentación con expresiones: el *Code Pad*

En las secciones anteriores hemos visto varias expresiones que nos permiten realizar distintos cálculos, como por ejemplo el cálculo `total + price` en la máquina expendedora y la expresión `name.substring(0,4)` en la clase `Student`.

En el resto del libro nos encontraremos con muchas más expresiones de este tipo, que en ocasiones se escriben con símbolos de operadores (como "+") y en otros casos se escriben como llamadas a métodos (como por ejemplo `substring`). Cuando nos encontremos con nuevos operadores y métodos, a menudo será útil probar diferentes ejemplos para ver cómo funcionan.

El Code Pad (teclado de código), que ya hemos utilizado brevemente en el Capítulo 1, nos puede ayudar a experimentar con las expresiones Java (Figura 2.6). En él, podemos escribir expresiones, que serán evaluadas inmediatamente mostrándose los correspondientes resultados. Esto es muy útil para probar nuevos operadores y métodos.

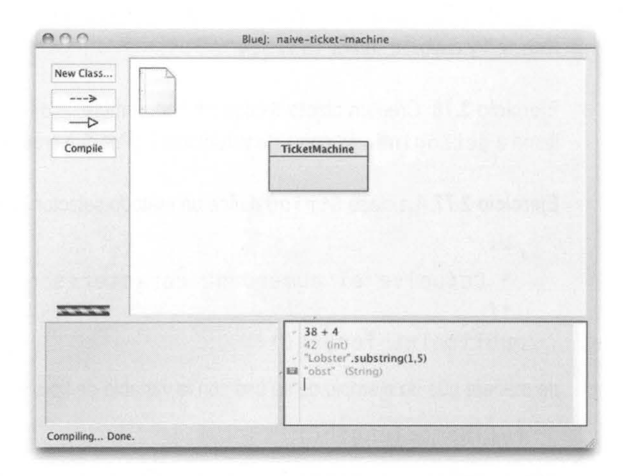


Figura 2.6
El Code Pad de BlueJ.

Ejercicio 2.79 Considere las siguiente expresiones. Intente predecir los resultados y luego escríbalas en el Code Pad para comprobar sus respuestas.

```
99 + 3
"cat" + "fish"
"cat" + 9
9 + 3 + "cat"
"cat" + 3 + 9
"catfish".substring(3,4)
"catfish".substring(3,8)
```

¿Ha aprendido algo que no esperara en este ejercicio? En caso afirmativo, ¿qué es lo que ha aprendido?

Cuando el resultado de una expresión en el Code Pad es un objeto (como por ejemplo `String`), estará marcado con un pequeño símbolo rojo de objeto. Puede hacer doble clic sobre este símbolo para inspeccionarlo o arrastrarlo al banco de objetos para utilizarlo más adelante. También puede declarar variables y escribir instrucciones completas en el Code Pad.

Cuando se encuentre con nuevos operadores y llamadas a métodos, suele ser buena idea probarlos aquí para tratar de familiarizarse con su comportamiento.

También puede explorar el uso de las variables en el Code Pad. Pruebe lo siguiente:

```
sum = 99 + 3;
```

Aparecerá el siguiente mensaje de error:

```
Error: cannot find symbol - variable sum
```

Esto se debe a que Java requiere que a todas las variables (`sum`, en este caso) se les asigne un tipo antes de utilizarlas. Recuerde que cada vez que hemos introducido por primera vez en el código fuente un campo, un parámetro o una variable local, hemos añadido delante del nombre un tipo, como por ejemplo `int` o `String`. Teniendo esto en cuenta, ahora pruebe a hacer lo siguiente en el Code Pad:

```
int sum = 0;
sum = 99 + 3;
```

Esta vez no se produce ningún error, porque `sum` ha sido definida con un tipo y puede utilizarse sin necesidad de repetir el tipo posteriormente. Si a continuación escribimos

```
sum
```

en una línea sin ninguna otra información (y sin punto y coma), podremos ver el valor que tiene almacenado actualmente esa variable.

Ahora pruebe esto en el Code Pad:

```
String swimmer = "cat" + "fish";
swimmer
```

De nuevo, hemos asignado un tipo apropiado a la variable `swimmer`, lo que nos permite asignarle un valor y consultar cuál es el valor almacenado. Esta vez, hemos decidido asignarle el valor deseado al mismo tiempo que declarábamos la variable.

¿Cuál esperaría que fuera el resultado si hacemos lo siguiente?

```
String fish = swimmer;
fish
```

Pruébelo. ¿Qué cree que ha pasado en esta asignación?

Ejercicio 2.80 Abra el Code Pad en el proyecto *better-ticket-machine*. Escriba lo siguiente:

```
TicketMachine t1 = new TicketMachine(1000);
t1.getBalance()
t1.insertMoney(500);
t1.getBalance()
```

Asegúrese de escribir estas líneas exactamente como aparecen aquí; preste atención en especial al hecho de si hay un punto y coma al final de la línea o no. Observe lo que devuelven en cada caso las llamadas a `getBalance`.

Ejercicio 2.81 Ahora añada lo siguiente en el Code Pad:

```
TicketMachine t2 = t1;
```

¿Qué esperaría que devuelva una llamada a `t2.getBalance()`? Pruébelo.

Ejercicio 2.82 Añada lo siguiente:

```
t1.insertMoney(500);
```

¿Qué esperaría que devuelva la siguiente instrucción? Piense cuidadosamente acerca de esto antes de comprobarlo y asegúrese de utilizar esta vez la variable `t2`.

```
t2.getBalance()
```

¿Ha obtenido la respuesta que esperaba? ¿Puede encontrar una conexión entre las variables `t1` y `t2` que explique lo que está sucediendo?

2.23 Resumen

En este capítulo hemos cubierto los fundamentos de cómo crear una definición de clase. Las clases contienen campos, constructores y métodos que definen el estado y el comportamiento de los objetos. Dentro del cuerpo de un constructor o método, una secuencia de instrucciones implementa la parte correspondiente de su comportamiento. Las variables locales pueden utilizarse como espacio almacenamiento de datos temporal, como ayuda para implementar el comportamiento requerido. Hemos visto las instrucciones de asignación y las asignaciones condicionales y en posteriores capítulos iremos añadiendo otros tipos de instrucciones.

Términos introducidos en el capítulo

campo, variable de instancia, constructor, método, cabecera del método, cuerpo de método, parámetro real, parámetro formal, selector, mutador, declaración, inicialización, bloque, instrucción, instrucción de asignación, instrucción condicional, instrucción return, tipo de retorno, comentario, expresión, operador, variable, variable local, ámbito, tiempo de vida

Resumen de conceptos

- **creación de objetos** Algunos objetos no pueden construirse a menos que proporcionemos información adicional.
- **campo** Los campos almacenan datos que un objeto tiene que utilizar. Los campos se conocen también con el nombre de variables de instancia.
- **comentario** Los comentarios se insertan en el código fuente de una clase para proporcionar explicaciones a los lectores humanos. No tienen ningún efecto sobre la funcionalidad de la clase.
- **constructor** Los constructores permiten configurar cada objeto apropiadamente en el momento de crearlo por primera vez.
- **ámbito** El ámbito de una variable define la sección del código fuente desde la que se puede acceder a esa variable.
- **tiempo de vida** El tiempo de vida de una variable describe durante cuánto tiempo continúa existiendo la variable antes de ser destruida.
- **asignación** Las instrucciones de asignación almacenan el valor representado por el lado derecho de la instrucción en la variable especificada a la izquierda.

- **método selector** Los métodos selectores devuelven información acerca del estado de un objeto.
- **método mutador** Los métodos mutadores cambian el estado de un objeto.
- **println** El método `System.out.println` imprime su parámetro en el terminal de texto.
- **condicional** Una instrucción condicional lleva a cabo una de dos posibles acciones basándose en el resultado de una prueba.
- **expresión booleana** Las expresiones booleanas solo tienen dos posibles valores: verdadero (true) y falso (false). Se las utiliza a menudo a la hora de controlar la elección entre las dos rutas de ejecución especificadas en una instrucción condicional.
- **variable local** Una variable local es una variable declarada y utilizada dentro un único método. Su ámbito y su tiempo de vida están limitados a los del propio método.

Los siguientes ejercicios están diseñados para ayudarle a experimentar con los conceptos de Java que hemos presentado en este capítulo. Tendrá que crear sus propias clases que contengan elementos tales como campos, constructores, métodos, instrucciones de asignación e instrucciones condicionales.

Ejercicio 2.83 A continuación se muestra el esbozo de una clase `Book`, que puede encontrar en el proyecto *book-exercise*. El esbozo presentado ya define dos campos y un constructor que se utiliza para inicializar los campos. En este y en los siguientes ejercicios tendrá que ir añadiendo características a este esbozo de clase que emplearemos para representar libros.

Añada dos métodos selectores a la clase –`getAuthor` y `getTitle`– que devuelvan como resultado los campos `author` y `title`, respectivamente, para saber cuál es el autor y el título de un libro. Compruebe la clase creando algunas instancias e invocando los métodos.

```
/**
 * Una clase que mantiene información sobre un libro.
 * Puede formar parte de una aplicación mayor, como por
 * ejemplo un sistema de control de biblioteca.
 *
 * @author (Introduzca aquí su nombre.)
 * @version (Introduzca aquí la fecha de hoy.)
 */
public class Book
{
    // Los campos.
    private String author;
    private String title;

    /**
     * Configurar los campos author y title en el momento de
     * construir este objeto.
     */
```

```
    public Book(String bookAuthor, String bookTitle)
    {
        author = bookAuthor;
        title = bookTitle;
    }

    // Añadir aquí los métodos...
}
```

Ejercicio 2.84 Añada dos métodos, `printAuthor` y `printTitle`, a la clase `Book`. Estos métodos deben imprimir los campos `author` y `title`, respectivamente, en la ventana de terminal.

Ejercicio 2.85 Añada un campo, `pages`, a la clase `Book` para almacenar el número de páginas del libro. Deberá ser de tipo `int`, y su valor inicial debe pasarse al único constructor existente, junto con las cadenas de caracteres correspondientes a `author` y `title`. Incluya un método selector `getPages` apropiado para este campo.

¿Son inmutables los objetos `Book` que ha implementado? Justifique su respuesta.

Ejercicio 2.86 Añada un método, `printDetails`, a la clase `Book`. Este método debe imprimir los detalles relativos al autor, el título y el número de páginas en la ventana de terminal. El formateo de los detalles queda a su elección. Por ejemplo, podrían mostrarse los tres elementos en una misma línea, o cada uno de ellos en una línea separada. También puede incluir, si lo desea, un texto explicativo, para que el usuario identifique fácilmente cuál es el autor y cuál es el título, como por ejemplo:

```
Title: Robinson Crusoe, Author: Daniel Defoe, Pages: 232
```

Ejercicio 2.87 Añada un campo adicional, `refNumber`, a la clase `Book`. Este campo podría almacenar, por ejemplo, un número de referencia para una biblioteca. Tiene que ser de tipo `String` y hay que inicializarlo con la cadena de caracteres de longitud cero (`""`) en el constructor, ya que su valor inicial no se pasa mediante ningún parámetro del constructor. En lugar de ello, defina un mutador para ese campo con la siguiente cabecera:

```
public void setRefNumber(String ref)
```

El cuerpo de este método debe asignar el valor del parámetro al campo `refNumber`. Añada el correspondiente método selector `getRefNumber` para poder comprobar fácilmente que el mutador funciona de forma correcta.

Ejercicio 2.88 Modifique su método `printDetails` para que se imprima también el número de referencia. No obstante, el método debe imprimir el número de referencia solo si ha sido configurado –es decir, si la cadena `refNumber` tiene una longitud distinta de cero. Si no ha sido configurado, entonces imprima en su lugar la cadena `"ZZZ"`. *Sugerencia:* utilice una instrucción condicional cuya comprobación invoque al método `length` para la cadena de caracteres `refNumber`.

Ejercicio 2.89 Modifique su método mutador `setRefNumber` para que configure el campo `refNumber` solo si el parámetro es una cadena de al menos tres caracteres. Si tiene menos de tres caracteres, entonces imprima un mensaje de error y deje el campo sin modificar.

Ejercicio 2.90 Añada un campo entero adicional, `borrowed`, a la clase `Book`. Este campo llevará la cuenta del número de veces que se ha tomado prestado un libro. Añada un mutador, `borrow`, a la clase.

Este método debe incrementar el campo en una unidad cada vez que sea invocado. Incluya un selector, `getBorrowed`, que devuelva como resultado el valor de este nuevo campo. Modifique `printDetails` para incluir el valor de este campo, junto con un texto explicativo.

Ejercicio 2.91 Añada un campo de tipo `boolean` adicional, `courseText`, a la clase `Book`. Este campo indica si el libro se está utilizando como libro de texto de un curso o no. El campo debe configurarse mediante un parámetro del constructor y ese campo es immutable. Proporcione un método selector para el mismo denominado `isCourseText`.

Ejercicio 2.92 *Ejercicio avanzado.* Cree un proyecto nuevo, *heater-exercise*, en BlueJ. Edite los detalles en la descripción del proyecto –la nota de texto que se ve en el diagrama. Cree una clase, `Heater`, que represente a un calefactor y que contenga un único campo, `temperature`, cuyo tipo sea de coma flotante de doble precisión (*double-precision floating point*) –consulte en la Sección B.1 del Apéndice B el nombre del tipo Java que se corresponde con esta descripción. Defina un constructor que no admita ningún parámetro. El campo `temperature` debe configurarse con el valor 15.0 en el constructor. Defina los mutadores `warmer` y `cooler`, cuyo efecto consiste en incrementar o reducir el valor de la temperatura en 5.0°, respectivamente. Defina un método selector para devolver el valor de `temperature`.

Ejercicio 2.93 *Ejercicio avanzado.* Modifique su clase `Heater` para definir tres nuevos campos de tipo coma flotante de doble precisión: `min`, `max` e `increment`. Los valores de `min` y `max` deben configurarse mediante parámetros pasados al constructor. El campo `increment` debe inicializarse con el valor 5.0 en el constructor. Modifique las definiciones de `warmer` y `cooler` para que empleen el valor de `increment` en lugar de un valor explícito de 5.0. Antes de continuar con este ejercicio compruebe que todo funciona como antes.

Ahora, modifique el método `warmer` para que no permita configurar la temperatura a un valor mayor que el indicado por `max`. De forma similar, modifique `cooler` para que no permita que la temperatura se configure con un valor menor que `min`. Compruebe que la clase funciona adecuadamente. Ahora añada un método `setIncrement`, que admita un único parámetro de tipo apropiado y que lo utilice para configurar el valor de `increment`. Una vez más, compruebe que la clase funciona como cabría esperar, creando algunos objetos `Heater` dentro de BlueJ. ¿Siguen funcionando las cosas como se esperaría si pasamos un valor negativo al método `setIncrement`? Añada una comprobación a este método para impedir que se asigne a `increment` un valor negativo.

CAPÍTULO

3

Interacción de objetos

Principales conceptos explicados en el capítulo:

- abstracción
- modularización
- creación de objetos
- diagramas de objetos
- llamadas a métodos
- depuradores

Estructuras Java explicadas en este capítulo:

tipos de clases, operadores lógicos (&&, ||), concatenación de cadenas, operador módulo (%), construcción de objetos (`new`), llamadas a métodos (notación de punto), `this`

En los capítulos anteriores hemos examinado qué son los objetos y cómo se implementan. En concreto, hemos hablado de los campos, los constructores y los métodos al examinar las definiciones de clases.

Ahora vamos a ir un paso más allá. Para construir aplicaciones interesantes, no basta con construir objetos que funcionen de manera individual. Lo que hay que hacer es combinar los objetos con el fin de que puedan cooperar y llevar a cabo una tarea común. En este capítulo, construiremos una pequeña aplicación a partir de tres objetos y veremos cómo hacer que los métodos invoquen a otros métodos para conseguir sus objetivos.

3.1 El ejemplo del reloj

El proyecto que utilizaremos para hablar de la interacción entre objetos es una pantalla para un reloj digital. La pantalla muestra las horas y los minutos, separados por un carácter de dos puntos (Figura 3.1). Para este ejercicio, construiremos primero un reloj con una visualización de 24 horas, al estilo europeo. Por tanto, la pantalla mostrará la hora desde 00:00 (medianoche) hasta 23:59 (un minuto antes de la medianoche). Al realizar una inspección más detallada, se descubre que construir un reloj de 12 horas es ligeramente más difícil que construir un reloj de 24 horas; por ello, dejaremos esa tarea para el final de este capítulo.

Figura 3.1
Pantalla de un reloj digital.

11:03

3.2 Abstracción y modularización

Una primera idea podría ser implementar toda la pantalla del reloj mediante una única clase. Eso es, después de todo, lo que hemos visto hasta ahora: cómo construir clases para llevar a cabo una determinada tarea.

Sin embargo, enfocaremos el problema de forma ligeramente distinta. Veremos si podemos identificar subcomponentes en el problema que podamos transformar en clases separadas. La razón principal para actuar así es la *complejidad.* A medida que avancemos en el libro, los ejemplos que utilicemos y los programas que construyamos serán cada vez más complejos. Las tareas triviales, como la de la máquina expendedora pueden resolverse como un único problema: es decir, podemos examinar la tarea completa y desarrollar una solución empleando una única clase. Sin embargo, para problemas más complejos, este enfoque es demasiado simplista. A medida que la complejidad de un problema aumenta, cada vez se hace más difícil controlar todos los detalles simultáneamente.

Concepto:

La **abstracción** es la capacidad de ignorar los detalles de las distintas partes, para centrar la atención en un nivel superior de un problema.

La solución que usaremos para tratar con el problema de la complejidad es la *abstracción.* Dividiremos el problema en una serie de subproblemas, que a su vez dividiremos en sub-subproblemas, y así sucesivamente, hasta que cada problema individual sea lo suficientemente pequeño como para poder resolverlo de manera sencilla. Una vez resuelto uno de los subproblemas, ya no dedicaremos más tiempo a pensar en los detalles de esa parte, sino que trataremos la solución como si fuera un único bloque componente que podemos emplear para solucionar el siguiente problema. Esta técnica se denomina en ocasiones *divide y vencerás.*

Expliquemos esto con un ejemplo. Imagine a dos ingenieros de una empresa automovilística diseñando un nuevo vehículo. Pueden pensar en las distintas características del vehículo, como por ejemplo la forma del cuerpo exterior, el tamaño y la ubicación del motor, el número y el tamaño de los asientos en el interior del coche, la separación exacta de las ruedas, etc. Por su parte, otro ingeniero cuyo trabajo sea diseñar el motor (bueno, en realidad, de eso se encarga un equipo completo de ingenieros, pero vamos a simplificar aquí las cosas un poquito para clarificar el ejemplo) se dedica a pensar en las distintas partes que tiene un motor: los cilindros, el mecanismo de inyección, el carburador, la electrónica, etc. Pensará en el motor no como si fuera una única entidad, sino como en una obra compleja compuesta de múltiples partes. Una de esas partes podría ser una bujía.

Por tanto, habrá un ingeniero (quizá en una empresa distinta) que diseñe bujías. Ese ingeniero pensará en una bujía como si fuera un artefacto complejo, formado por múltiples partes. Es posible que haya hecho complejos estudios para determinar exactamente qué tipo de metal utilizar para los contactos o qué clase de material y de proceso de producción emplear para el aislamiento.

Lo mismo vale para muchas otras partes del vehículo. Un diseñador en el nivel más alto considerará una rueda como si fuera un único componente. Otro ingeniero situado mucho más abajo en la cadena de producción podría dedicar su tiempo a pensar en la composición

química necesaria para obtener los materiales correctos con los que fabricar las llantas. Para el ingeniero de llantas, la llanta es algo muy complejo. La empresa de automóviles se limitará a comprar las llantas a la empresa que las vende y luego considerará cada llanta como una única entidad. Esto es lo que se llama abstracción.

El ingeniero de la empresa automovilística *se abstrae de* los detalles de la fabricación de la llanta, para poder concentrarse en los detalles de la construcción, por ejemplo, de la rueda. El diseñador que piensa en la forma del chasis del vehículo se abstrae de los detalles técnicos de las ruedas y del motor, para concentrarse en el diseño del chasis (lo único que le interesará saber es el tamaño del motor y de las ruedas).

Lo mismo vale para cualquier otro componente. Mientras que alguien puede estar preocupado por diseñar el espacio interior del vehículo donde se alojarán los pasajeros, alguna otra persona puede estar trabajando en desarrollar el tejido que terminará utilizándose para cubrir los asientos.

Lo importante es que, si se contempla con el suficiente detalle, un vehículo está compuesto de tantas partes distintas que es imposible que una única persona conozca todos los detalles de todas esas partes al mismo tiempo. Si fuera necesario que alguien conociera todos los detalles, jamás podríamos llegar a construir ni un solo vehículo.

Concepto:

La **modularización** es el proceso de dividir un todo en partes bien definidas que puedan construirse y examinarse por separado y que interactúen de formas bien definidas.

La razón por la que podemos llegar a construir vehículos es que los ingenieros utilizan la *modularización* y la abstracción. Dividen el vehículo en módulos independientes (rueda, motor, caja de cambios, asiento, volante, etc.) y encargan a diferentes personas que trabajen en los distintos módulos de forma independiente. Una vez construido un módulo, utilizan la abstracción: contemplan dicho módulo como un único componente que se emplea para construir componentes más complejos.

Por tanto, la modularización y la abstracción se complementan entre sí. La modularización es el proceso de dividir grandes cosas (problemas) en partes más pequeñas, mientras que la abstracción es la capacidad de ignorar los detalles para centrarse en la panorámica general.

3.3 Abstracción en el software

Los mismos principios de modularización y abstracción de los que hemos hablado en la sección anterior se emplean en el desarrollo de software. Para ayudarnos a mantener una visión panorámica en los problemas complejos, tratamos de identificar subcomponentes que podamos programar como entidades independientes. Después, tratamos de usar esos subcomponentes como si fueran partes simples, sin preocuparnos acerca de su complejidad interna.

En la programación orientada a objetos, estos componentes y subcomponentes son precisamente objetos. Si estuviéramos tratando de construir un vehículo en software, utilizando un lenguaje orientado a objetos, trataríamos de hacer lo que hacen los ingenieros de diseño de vehículos. En lugar de implementar el automóvil como un único objeto monolítico, construiríamos primero objetos separados para el motor, la caja de cambios, la rueda, el asiento, etc., y luego ensamblaríamos el objeto vehículo a partir de esos objetos más pequeños.

Identificar qué tipo de objetos (y por tanto de clases) hay que incluir en un sistema software para cualquier problema dado no siempre es fácil, y tendremos que hablar largo y tendido sobre ello posteriormente en el libro. Por ahora, comencemos con un ejemplo relativamente simple y volvamos nuestra atención al reloj digital.

3.4 Modularización en el ejemplo del reloj

Echemos un vistazo más detallado al ejemplo de la pantalla del reloj. Utilizando los conceptos de abstracción que acabamos de describir, queremos encontrar la mejor forma de contemplar este ejemplo para poder escribir algunas clases que lo implementen. Una forma de contemplarlo consiste en considerarlo compuesto por una única pantalla con cuatro dígitos (dos dígitos para las horas y otros dos para los minutos). Si ahora nos abstraemos de esa visión de muy bajo nivel, podemos ver que el reloj también podría contemplarse como formado por dos pantallas separadas de dos dígitos (una pareja para las horas y otra para los minutos). Una de las parejas comienza en 0, se incrementa en 1 cada hora y vuelve a 0 después de alcanzar su límite de 23. La otra pareja vuelve a 0 después de alcanzar su límite de 59. La similitud en el comportamiento de estas dos pantallas podría entonces llevarnos a abstraernos aun más, evitando contemplar de manera distinta la pantalla de las horas y la de los minutos. En lugar de ello, podríamos pensar en esas dos pantallas como objetos capaces de mostrar valores que van desde cero a un determinado límite. El valor puede incrementarse, pero si alcanza el límite, vuelve a cero. Con esto parece que hemos alcanzado un nivel apropiado de abstracción, que podemos representar mediante una clase: una clase para una pantalla de dos dígitos.

Para nuestra pantalla del reloj, primero programaremos una clase para una pantalla numérica de dos dígitos (Figura 3.2) y luego la dotaremos de un método selector para consultar su valor y de dos métodos mutadores para fijar el valor e incrementarlo. Una vez definida esta clase, podemos simplemente crear dos objetos de esa clase con diferentes límites, para construir la pantalla completa del reloj.



Figura 3.2
Una pantalla numérica de dos dígitos.

3.5 Implementación de la pantalla del reloj

Como hemos dicho, para construir la pantalla del reloj, primero crearemos una pantalla numérica de dos dígitos. Esta pantalla necesita almacenar dos valores. Uno de ellos es el límite hasta el que puede contar antes de volver a cero; el otro será el valor actual. Representaremos ambos valores como campos enteros en nuestra clase (Código 3.1).

Código 3.1
Clase para la pantalla numérica de dos dígitos.

```
public class NumberDisplay
{
    private int limit;
    private int value;
    Se omiten el constructor y los métodos.
}
```

Examinaremos posteriormente los detalles restantes de esta clase. Pero antes de eso, vamos a asumir que podemos construir la clase `NumberDisplay` y vamos a reflexionar un poco más sobre la pantalla completa del reloj. Podríamos construir una pantalla completa de reloj mediante un objeto que tenga, internamente, dos pantallas numéricas (una para las horas y otra para los minutos). Cada una de las pantallas numéricas sería un campo dentro de la pantalla del reloj completo (Código 3.2). Aquí, vamos a hacer uso de un detalle que no habíamos mencionado anteriormente: las *clases definen tipos*.

Código 3.2
La clase `ClockDisplay` que contiene dos campos `NumberDisplay`.

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Se omiten el constructor y los métodos.
}
```

Concepto:

Las **clases definen tipos.** Un nombre de clase puede utilizarse como tipo para una variable. Las variables que tienen una clase como tipo pueden almacenar objetos de dicha clase.

Cuando hemos hablado de los campos en el Capítulo 2, hemos dicho que la palabra "private" en la declaración del campo va seguida por un tipo y un nombre de campo. Aquí hemos utilizado la clase `NumberDisplay` como tipo para los campos denominados `hours` y `minutes`. Esto demuestra que los nombres de clases pueden utilizarse como tipos.

El tipo de un campo especifica qué especie de valores pueden almacenarse en ese campo. Si el tipo es una clase, el campo podrá contener objetos de dicha clase.

3.6 Diagramas de clases y diagramas de objetos

La estructura descrita en la sección anterior (un objeto `ClockDisplay` que contiene dos objetos `NumberDisplay`) puede visualizarse mediante un *diagrama de objetos* como el que se muestra en la Figura 3.3a. En este diagrama vemos que estamos tratando con tres objetos. La Figura 3.3b muestra el *diagrama de clases* para la misma situación.

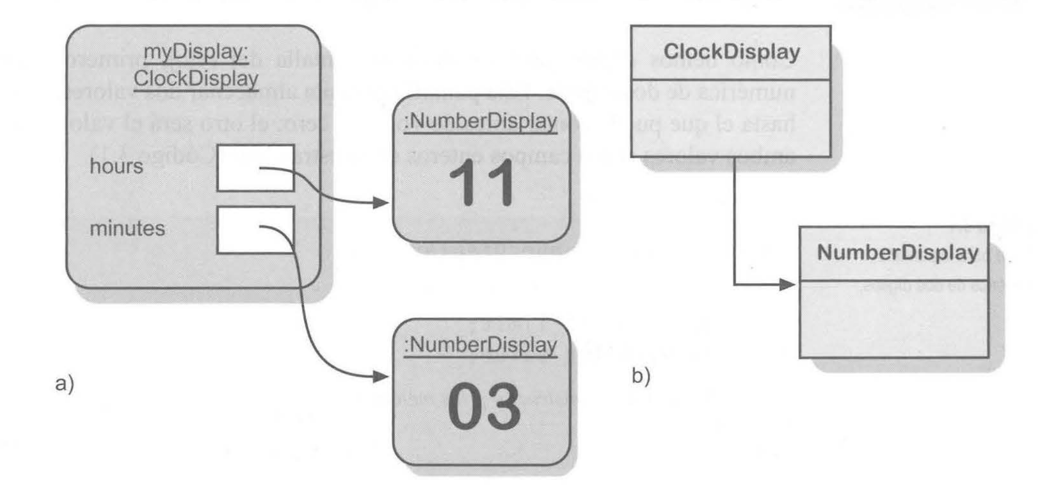


Figura 3.3
Diagrama de objetos y diagrama de clases para `ClockDisplay`.

Concepto:

El **diagrama de clases** muestra las clases de una aplicación y las relaciones entre ellas. Proporciona información acerca del código fuente y presenta una vista estática de un programa.

Concepto:

El **diagrama de objetos** muestra los objetos y sus relaciones en un instante determinado durante la ejecución de una aplicación. Proporciona información acerca de los objetos en tiempo de ejecución y presenta una vista dinámica de un programa.

Observe que el diagrama de clases solo muestra dos clases, mientras que en el diagrama de objetos aparecen tres objetos distintos. Esto tiene que ver con el hecho de que se pueden crear varios objetos a partir de una misma clase. En este caso, creamos dos objetos `NumberDisplay` a partir de la clase `NumberDisplay`.

Estos dos diagramas ofrecen visiones distintas de la misma aplicación. El diagrama de clases muestra la *vista estática*. En él se refleja lo que tenemos en el momento de escribir el programa. Tenemos dos clases y la flecha indica que la clase `ClockDisplay` hace uso de la clase `NumberDisplay` (`NumberDisplay` aparece mencionado en el código fuente de `ClockDisplay`). También vemos, por eso, que `ClockDisplay` *depende de* `NumberDisplay`.

Para comenzar el programa, crearemos un objeto de clase `ClockDisplay`. Programaremos la pantalla del reloj para que esta cree automáticamente dos objetos `NumberDisplay`, con el fin de utilizarlos ella misma. Por tanto, el diagrama de objetos muestra la situación en *tiempo de ejecución* (cuando se está ejecutando la aplicación). Esto se denomina también *vista dinámica.*

Concepto:

Referencias a objetos. Las variables con un **tipo de objeto** almacenan referencias a objetos.

El diagrama de objetos también muestra otro detalle importante: cuando una variable almacena un objeto, el objeto no se almacena directamente en la variable, sino que lo que la variable contiene es una *referencia a objeto.* En el diagrama, la variable se muestra mediante un recuadro blanco, y la referencia a objeto aparece como una flecha. El objeto al que se hace referencia está almacenado fuera del objeto en el que la referencia aparece, y es precisamente esa referencia a objeto lo que enlaza los dos objetos entre sí.

Es muy importante comprender estos dos diagrama distintos, que representan dos visiones diferentes de la aplicación. BlueJ proporciona solo la vista estática. Podemos ver el diagrama de clases en su ventana principal. Para planificar y comprender programas Java, necesitamos ser capaces de construir diagramas de objetos sobre papel o en nuestra cabeza. Cuando pensemos acerca de lo que nuestro programa va a hacer, pensaremos en las estructuras de objetos que creará y en cómo van a interactuar esos objetos. Ser capaz de visualizar las estructuras de objetos es esencial.

Ejercicio 3.1 Piense de nuevo en el proyecto *lab-classes* que hemos visto en el Capítulo 1 y el Capítulo 2. Imagine que creamos un objeto `LabClass` y tres objetos `Student`. Imagine también que después matriculamos a los tres estudiantes en este laboratorio. Trate de dibujar un diagrama de clases y un diagrama de objetos para dicha situación. Identifique y explique las diferencias existentes entre ambos diagramas.

Ejercicio 3.2 ¿En qué momento o momentos puede cambiar un diagrama de clases? ¿Cómo se modifica?

Ejercicio 3.3 ¿En qué momento o momentos puede cambiar un diagrama de objetos? ¿Cómo se modifica?

Ejercicio 3.4 Escriba una definición de un campo denominado `tutor` que pueda almacenar una referencia a un objeto de tipo `Instructor`, para poder definir que un determinado profesor es un tutor.

3.7 Tipos primitivos y tipos de objeto

Java trabaja con dos especies muy distintas de tipos: *tipos primitivos* y *tipos de objeto.* Los tipos primitivos están todos ellos predefinidos en el lenguaje Java. Entre ellos se incluyen

Concepto:

Los **tipos primitivos** en Java son los tipos que no son de objeto. Los tipos primitivos más comunes son `int`, `boolean`, `char`, `double` y `long`. Los tipos primitivos no disponen de métodos.

`int` y `boolean`. En el Apéndice B se proporciona una lista completa de tipos primitivos. Los tipos de objeto son aquellos que están definidos mediante clases. Algunas clases se definen mediante el sistema Java estándar (como por ejemplo `String`); otras clases son las que escribimos nosotros mismos.

Tanto los tipos primitivos como los tipos de objeto pueden emplearse como tipos, pero hay situaciones en las que se comportan de forma diferente. Una de las diferencias afecta al modo en que se almacenan los valores. Como hemos podido ver en nuestros diagramas, los valores primitivos se almacenan directamente en una variable (escribimos el valor directamente en el recuadro de la variable –por ejemplo, en el Capítulo 2, Figura 2.3). Por el contrario, los objetos no se almacenan directamente en la variable, sino que lo que se almacena es una referencia al objeto (dibujada como una flecha en nuestros diagramas, como por ejemplo en la Figura 3.3a).

Veremos posteriormente otras diferencias entre tipos primitivos y tipos de objeto.

3.8 El código fuente para `ClockDisplay`

Antes de comenzar a analizar el código fuente, será útil que echemos un vistazo al ejemplo.

Ejercicio 3.5 Inicie BlueJ, abra el ejemplo *clock-display* y experimente con él. Para ello, cree un objeto `ClockDisplay` utilizando el constructor que no admite ningún parámetro y luego abra una ventana del inspector para este objeto. Con el inspector abierto, invoque los métodos del objeto. Observe el campo `displayString` en el inspector. Lea el comentario del proyecto (haciendo doble clic sobre el icono de nota de texto en la pantalla principal) para obtener más información.

3.8.1 Clase `NumberDisplay`

Vamos ahora a analizar una implementación completa de esta tarea. La solución está contenida en el proyecto *clock-display* de los ejemplos proporcionados con este libro. En primer lugar, examinaremos la implementación de la clase `NumberDisplay`. El Código 3.3 muestra el código fuente completo. En conjunto, esta clase es bastante sencilla. Tiene los dos campos de los que hemos hablado anteriormente (Sección 3.5), un constructor y cuatro métodos (`getValue`, `setValue`, `getDisplayValue` e `increment`).

El constructor recibe como parámetro el límite de recuento de la pantalla. Por ejemplo, si se pasa 24 como límite de recuento, la pantalla volverá a 0 al alcanzar dicho valor. Por tanto, el rango de visualización de la pantalla del reloj irá de 0 a 23. Esta característica nos permite utilizar esta clase para visualizar tanto horas como minutos. Para la visualización de horas crearemos un `NumberDisplay` con un límite de 24; para la visualización de los minutos, crearemos otro con límite de 60.

El constructor almacena entonces el límite de recuento en un campo e inicializa con 0 el valor actual de la pantalla.

A continuación, sigue un método selector simple para el valor de visualización actual (`getValue`). Este método permite a los otros objetos leer el valor actual de la pantalla.

Código 3.3
Implementación de la clase NumberDisplay.

```
/**
 * La clase NumberDisplay representa una pantalla numérica digital
 * que puede almacenar valores comprendidos entre cero y un
 * determinado límite.
 * El límite puede especificarse a la hora de crear la pantalla.
 * Los valores van de 0 (incluido) hasta límite-1. Por ejemplo,
 * si se usa para los segundos en un reloj digital, el límite
 * sería 60, lo que nos daría un rango de visualización
 * comprendido entre 0 y 59.
 * Al incrementarse, la pantalla vuelve automáticamente a cero
 * cuando alcanza el límite.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 2011.07.31
 *
 */
public class NumberDisplay
{
    private int limit;
    private int value;

    /**
     * Constructor para los objetos de la clase NumberDisplay
     */
    public NumberDisplay(int rollOverLimit)
    {
        limit = rollOverLimit;
        value = 0;
    }

    /**
     * Devuelve el valor actual.
     */
    public int getValue()
    {
        return value;
    }

    /**
     * Configura el valor de la pantalla con el nuevo valor
     * especificado. Si el nuevo valor es menor que cero o
     * superior al límite, no hace nada.
     */
    public void setValue(int replacementValue)
    {
        if((replacementValue >= 0) &&
                (replacementValue < limit)) {
```

Código 3.3
(continuación)
Implementación de la clase `NumberDisplay`.

```
            value = replacementValue;
        }

    /**
     * Devuelve el valor de visualización (es decir, el valor
     * actual en forma de String de dos dígitos. Si el valor es
     * menor que diez, lo rellena con un cero a la izquierda).
     */
    public String getDisplayValue()
    {
        if(value < 10) {
            return "0" + value;
        }
        else {
            return "" + value;
        }
    }

    /**
     * Incrementa el valor de visualización en una unidad,
     * volviendo a cero si se alcanza el límite.
     */
    public void increment()
    {
        value = (value + 1) % limit;
    }
}
```

El siguiente método mutador `setValue` es más interesante. Su definición es la siguiente:

```
public void setValue(int replacementValue)
{
    if((replacementValue >= 0) && (replacementValue < limit)) {
        value = replacementValue;
    }
}
```

Aquí, pasamos el nuevo valor para la pantalla como un parámetro del método. Sin embargo, antes de asignar el valor, tenemos que comprobar que es legal. El rango legal para el valor, como hemos dicho antes, va de 0 hasta una unidad por debajo del límite. Utilizamos una instrucción `if` para comprobar que el valor es legal antes de asignarlo. El símbolo "`&&`" es un operador "and" lógico, que hace que la condición de la instrucción `if` sea cierta si las dos condiciones a ambos lados del símbolo "`&&`" son ciertas. Puede ver más detalles en la nota sobre "Operadores lógicos" incluida a continuación. En el Apéndice C se muestra una tabla completa de los operadores lógicos de Java.

Operadores lógicos Los operadores lógicos actúan sobre valores booleanos (verdadero o falso) y producen un nuevo valor booleano como resultado. Los tres operadores lógicos más importantes son **and**, **or** y **not**. En Java se escriben de la forma siguiente:

&& (and)

|| (or)

! (not)

La expresión

```
a && b
```

es verdadera si tanto a como b son verdaderas y falsa en todos los casos restantes. La expresión

```
a || b
```

es verdadera si a o b o ambas son verdaderas y será falsa si las dos son falsas. La expresión

```
!a
```

es verdadera si a es falsa y será falsa si a es verdadera.

Ejercicio 3.6 ¿Qué sucede cuando se invoca al método `setValue` con un valor ilegal? ¿Es esta una buena solución? ¿Se le ocurre alguna solución mejor?

Ejercicio 3.7 Explique qué sucedería si sustituyéramos el operador ">=" de la prueba por el operador ">", de forma que quedará

```
if((replacementValue > 0) && (replacementValue < limit))
```

Ejercicio 3.8 Explique qué sucedería si sustituyéramos el operador && de la prueba por el operador ||, de forma que quedará

```
if((replacementValue >= 0) || (replacementValue < limit))
```

Ejercicio 3.9 ¿Cuál de las siguientes expresiones devuelve el valor *true*?

```
! (4 < 5)
! false
(2 > 2) || ((4 == 4) && (1 < 0))
(2 > 2) || (4 == 4) && (1 < 0)
(34 != 33) && ! false
```

Después de escribir sus respuestas en un papel, abra el Code Pad en BlueJ y haga la prueba. Verifique sus respuestas.

Ejercicio 3.10 Escriba una expresión utilizando las variables booleanas a y b que se evalúe como *true* cuando a y b sean ambas *true* o ambas *false*.

Ejercicio 3.11 Escriba una expresión utilizando las variables booleanas a y b que se evalúe como *true* cuando solo una de las dos, a o b, sea *true*, y que sea *false* si a y b are son ambas *false* o ambas *true*. (Esto se denomina también *or exclusiva*.)

Ejercicio 3.12 Considere la expresión (a && b). Escriba una expresión equivalente (una que se evalúe como *true* para exactamente los mismos valores de a y b) sin emplear el operador &&.

El siguiente método, `getDisplayValue`, también devuelve el valor de la pantalla, pero en un formato distinto. La razón es que queremos visualizar el valor en forma de cadena de dos dígitos. Es decir, si la hora actual es las 3:05, queremos que la pantalla muestre `03:05` y no `3:5`. Para poder hacer esto fácilmente, hemos implementado el método `getDisplayValue`. Este método devuelve el valor actual en forma de cadena de caracteres y añade un 0 al principio si el valor es menor que 10. He aquí la sección relevante del código:

```
if(value < 10) {
    return "0" + value;
}
else {
    return "" + value;
}
```

Observe que el cero (`"0"`) está escrito entre dobles comillas. De ese modo, hemos escrito la *cadena de caracteres (string)* 0, y no el *número entero* 0. Por tanto, la expresión

```
"0" + value
```

está "sumando" una cadena de caracteres y un entero (porque el tipo de `value` es `int`). El operador de suma representa otra vez, por tanto, una concatenación de cadenas, como hemos visto en la Sección 2.9. Antes de continuar, vamos a examinar más en detalle la concatenación de cadenas de caracteres.

3.8.2 Concatenación de cadenas de caracteres

El operador suma (+) tiene diferentes significados, dependiendo del tipo de sus operandos. Si ambos operandos son números, representa la suma algebraica como cabría esperar. Por tanto,

```
42 + 12
```

suma esos dos números y el resultado es 54. Sin embargo, si los operandos son cadenas de caracteres, entonces el significado del signo más es la concatenación de cadenas, y el resultado es una única cadena de caracteres compuesta por ambos operandos uno a continuación de otro. Por ejemplo, el resultado de la expresión

```
"Java" + "with BlueJ"
```

es la cadena de caracteres

```
"Javawith BlueJ"
```

Observe que el sistema no añade automáticamente un espacio entre las cadenas. Si queremos un espacio tenemos que incluirlo nosotros mismos en una de las dos cadenas.

Si uno de los operandos de una operación suma es una cadena y el otro no lo es, entonces automáticamente se convierte el otro operando a una cadena, realizándose después una concatenación. Por tanto,

```
"answer: " + 42
```

da como resultado la cadena

```
"answer: 42"
```

Esto es así para todos los tipos. Independientemente del tipo que se "sume" a una cadena, dicho tipo se convertirá automáticamente a una cadena y luego se concatenará.

Volvamos a nuestro código del método `getDisplayValue`. Por ejemplo, si `value` contiene 3, entonces la instrucción

```
return "0" + value;
```

devolverá la cadena `"03"`. En caso de que el valor sea mayor que 9, utilizamos un pequeño truco:

```
return "" + value;
```

Aquí, concatenamos el valor con una cadena vacía. El resultado es que el valor se convertirá en una cadena y no se le añadirá ningún otro carácter como prefijo. Estamos utilizando el operador suma con el único propósito de forzar la conversión del valor entero a un valor de tipo `String`.

Ejercicio 3.13 ¿Funciona el método `getDisplayValue` correctamente en todas las circunstancias? ¿Qué suposiciones se han hecho dentro de él? ¿Qué sucede si creamos una pantalla numérica con un límite de por ejemplo 800?

Ejercicio 3.14 Indique si existe alguna diferencia en el resultado de escribir

```
return value + "";
```

en lugar de

```
return "" + value;
```

dentro del método `getDisplayValue`?

3.8.3 El operador módulo

El último método de la clase `NumberDisplay` incrementa el valor de la pantalla en 1. El método se preocupa de devolver el valor a cero cuando se alcanza el límite:

```
public void increment()
{
    value = (value + 1) % limit;
}
```

Este método utiliza el operador *módulo* (`%`). El operador módulo calcula el resto de una división entera. Por ejemplo, el resultado de la división

```
27 / 4
```

se puede expresar mediante números enteros como

```
resultado = 6, resto = 3
```

El operador módulo devuelve simplemente el resto de dicha división. Por tanto, el resultado de la expresión `(27 % 4)` sería `3`.

Ejercicio 3.15 Explique el operador módulo. Es posible que necesite consultar más recursos (recursos en línea sobre el lenguaje Java, otros libros de Java, etc.) para conocer más detalles.

Ejercicio 3.16 ¿Cuál es el resultado de la expresión `(8 % 3)`?

Ejercicio 3.17 Pruebe la expresión `(8 % 3)` en el Code Pad. Pruebe con otros números. ¿Qué sucede cuando se utiliza el operador módulo con números negativos?

Ejercicio 3.18 ¿Cuáles son todos los posibles resultados de la expresión `(n % 5)`, donde `n` es una variable entera?

Ejercicio 3.19 ¿Cuáles son todos los posibles resultados de la expresión `(n % m)`, donde `n` y `m` son variables enteras?

Ejercicio 3.20 Explique detalladamente cómo funciona el método `increment`.

Ejercicio 3.21 Escriba de nuevo el método `increment` sin el operador módulo, utilizando una instrucción if. ¿Qué solución es mejor?

Ejercicio 3.22 Utilizando el proyecto *clock-display* en BlueJ, pruebe la clase `NumberDisplay` creando unos pocos objetos `NumberDisplay` e invocando sus métodos.

3.8.4 La clase `ClockDisplay`

Ahora que hemos visto cómo construir una clase que define una pantalla numérica de dos dígitos, examinaremos con más detalle la clase `ClockDisplay` –la clase que permitirá crear dos pantallas numéricas con el fin de crear una pantalla completa de reloj. El Código 3.4 proporciona el código fuente completo de la clase `ClockDisplay`.

Como hemos hecho con la clase `NumberDisplay`, explicaremos brevemente todos los campos, constructores y métodos.

Código 3.4
Implementación de la clase `ClockDisplay`.

```
/**
 * La clase ClockDisplay implementa una pantalla de reloj digital
 * para un reloj de 24 horas, estilo europeo. El reloj muestra
 * las horas y los minutos.
 * El rango del reloj es de 00:00 (medianoche) a 23:59 (un
 * minuto antes de la medianoche).
 *
 * La pantalla del reloj recibe "pulsos" (a través del método
 * timeTick) cada minuto y reacciona incrementando la pantalla.
 * Esto se hace de la forma habitual en los relojes: la hora se
 * incrementa cuando los minutos pasan de nuevo a cero.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 2011.07.31
 */
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private String displayString; // simula la pantalla real
```

Código 3.4
(continuación)
Implementación de la clase ClockDisplay.

```
    /**
     * Constructor para los objetos ClockDisplay. Este constructor
     * crea un nuevo reloj inicializado con 00:00.
     */
    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }
    /**
     * Constructor para los objetos ClockDisplay. Este constructor
     * crea un nuevo reloj inicializado con la hora especificada
     * mediante los parámetros.
     */
    public ClockDisplay(int hour, int minute)
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        setTime(hour, minute);
    }
    /**
     * Este método debe invocarse una vez cada minuto - hace
     * que la pantalla del reloj avance un minuto.
     */
    public void timeTick()
    {
        minutes.increment();
        if(minutes.getValue() == 0) { // ¡ha vuelto a cero!
            hours.increment();
        }
        updateDisplay();
    }
    /**
     * Fija la hora de la pantalla con la hora
     * y los minutos especificados.
     */
    public void setTime(int hour, int minute)
    {
        hours.setValue(hour);
        minutes.setValue(minute);
        updateDisplay();
    }
    /**
     * Devuelve la hora actual de esta pantalla en el
     * formato HH:MM.
     */
```

Código 3.4
(continuación)
Implementación de la clase `ClockDisplay`.

```
    public String getTime()
    {
        return displayString;
    }

    /**
     * Actualiza la cadena interna que representa la pantalla.
     */
    private void updateDisplay()
    {
        displayString = hours.getDisplayValue() + ":" +
                        minutes.getDisplayValue();
    }
}
```

En este proyecto, utilizamos el campo `displayString` para simular el dispositivo real de visualización del reloj (como hemos podido ver en el Ejercicio 3.5). Si este software se ejecutara en un reloj real, lo que haríamos en lugar de ello sería presentar la salida en la auténtica pantalla del reloj. Por tanto, esta cadena sirve como simulación software del dispositivo de salida del reloj.[1]

Para conseguir esto, utilizamos un campo de cadena y un método:

```
public class ClockDisplay
{
    private String displayString;

    Se omiten otros campos y métodos.

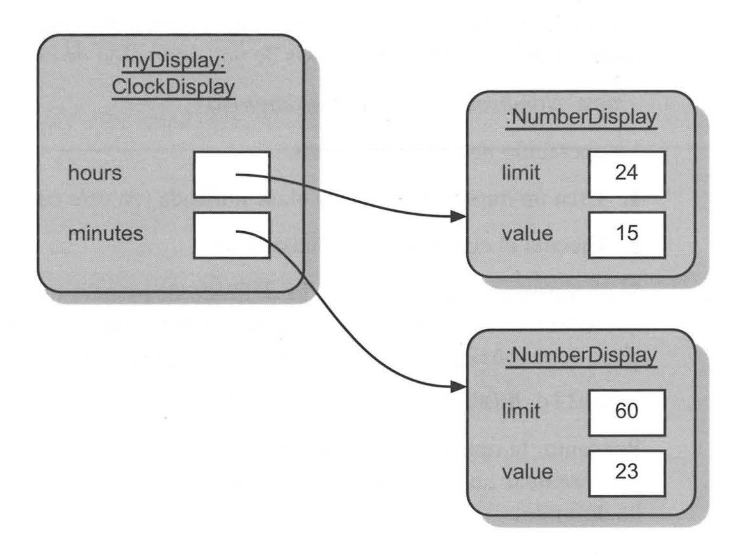
    /**
     * Actualiza la cadena interna que representa la pantalla.
     */
    private void updateDisplay()
    {
            Se omite la implementación del método.
    }
}
```

Cada vez que deseamos que cambie la visualización del reloj, tendremos que llamar al método interno `updateDisplay`. En nuestra simulación, este método modificará la cadena de visualización (más adelante examinaremos el código fuente utilizado para esto). En un reloj real, este método también existiría, aunque lo que allí haría sería modificar la pantalla real del reloj.

Además de la cadena de visualización, la clase `ClockDisplay` tiene otros dos campos más: `hours` y `minutes`. Cada uno de estos campos puede almacenar un objeto de tipo `NumberDisplay`. El valor lógico de la pantalla del reloj (la hora actual) está almacenado en esos objetos `NumberDisplay`. La Figura 3.4 muestra el diagrama de objetos de esta aplicación cuando la hora actual es 15:23.

[1] La carpeta de proyectos del libro incluye también una versión de este proyecto con una interfaz gráfica de usuario (GUI) simple, denominado *clock-display-with-GUI*. El lector curioso puede experimentar con este proyecto; no obstante, no vamos a explicarlo en este libro.

Figura 3.4
Diagrama de objetos de la pantalla del reloj.



3.9 Objetos que crean objetos

La primera cuestión que tenemos que plantearnos es: ¿de dónde vienen estos tres objetos? Cuando queramos utilizar una pantalla de reloj, podemos crear un objeto `ClockDisplay`. Después, asumimos que tiene horas y minutos. Por tanto, con solo crear una pantalla de reloj, esperamos haber creado implícitamente dos pantallas numéricas para las horas y los minutos.

Concepto:

Creación de objetos Los objetos pueden **crear** otros **objetos**, utilizando el operador `new`.

Como encargados de escribir la clase `ClockDisplay`, tenemos que asegurarnos de que esto suceda. Simplemente escribimos código en el constructor de `ClockDisplay` que se encargará de crear y almacenar dos objetos `NumberDisplay`. Puesto que el constructor se ejecuta automáticamente cada que vez que se crea un objeto `ClockDisplay`, los objetos `NumberDisplay` se crearán automáticamente al mismo tiempo. He aquí el código del constructor `ClockDisplay` que hace que esto funcione:

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;

    Se omiten los restantes campos.

    public ClockDisplay()
    {
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        updateDisplay();
    }

    Se omiten los métodos.
}
```

Cada una de las dos primeras líneas del constructor crea un nuevo objeto `NumberDisplay` y lo asigna a una variable. La sintaxis de una operación de creación de un nuevo objeto es:

new *NombreClase (lista-parámetros)*

La operación `new` hace dos cosas:

1. Crea un nuevo objeto de la clase indicada (en este caso, `NumberDisplay`).
2. Ejecuta el constructor de dicha clase.

Si el constructor de la clase se ha definido de manera que incluya parámetros, entonces habría que suministrar los parámetros reales en la instrucción `new`. Por ejemplo, el constructor de la clase `NumberDisplay` se definió para que espere recibir un parámetro entero:

```
public NumberDisplay (int rollOverLimit)
```

Por tanto, la operación `new` para la clase `NumberDisplay`, que invoca este constructor, deberá proporcionar un parámetro real de tipo `int` para ajustarse a la cabecera del constructor que se ha definido:

```
new NumberDisplay (24);
```

Esto es igual que para los métodos que hemos explicado en la Sección 2.4. Con este constructor, hemos conseguido lo que queríamos: si alguien crea ahora un objeto `ClockDisplay`, se ejecutará automáticamente el constructor de `ClockDisplay` y este creará dos objetos `NumberDisplay`. Con eso, la pantalla del reloj estará lista para funcionar.

Ejercicio 3.23 Cree un objeto `ClockDisplay` seleccionando el siguiente constructor:

```
new ClockDisplay()
```

Invoque su método `getTime` para averiguar la hora inicial con el que se ha configurado el reloj. ¿Puede explicar por qué el reloj parte de esa hora concreta?

Ejercicio 3.24 ¿Cuántas veces será necesario invocar el método `timeTick` en un objeto `ClockDisplay` recién creado para hacer que su hora alcance el valor 01:00? ¿De qué otra forma podría hacer que mostrara esa hora?

Ejercicio 3.25 Cree un objeto `NumberDisplay` con límite de 80 en el Code Pad escribiendo:

```
NumberDisplay nd = new NumberDisplay(80);
```

A continuación, invoque sus métodos `getValue()`, `setValue(int value)` e `increment()` en el Code Pad (por ejemplo, escribiendo `nd.getValue()`). Observe que las instrucciones (mutadores) necesitan un punto y coma al final, mientras que las expresiones (selectores) no lo necesitan.

Ejercicio 3.26 Escriba la signatura de un constructor que se corresponda con la siguiente instrucción de creación de un objeto:

```
new Editor("readme.txt", -1)
```

Ejercicio 3.27 Escriba instrucciones Java que definan una variable denominada `window` de tipo `Rectangle`, y luego cree un objeto rectángulo y asígnelo a dicha variable. El constructor para el rectángulo tiene dos parámetros `int`.

3.10 Constructores múltiples

Puede que haya observado, al crear un objeto `ClockDisplay`, que el menú emergente ofrece dos formas de hacerlo:

```
new ClockDisplay()
new ClockDisplay(int hour, int minute)
```

Concepto:

Sobrecarga. Una clase puede contener más de un constructor o más de un método con el mismo nombre, siempre que cada uno tenga un conjunto diferente de tipos de parámetros.

Esto se debe a que la clase `ClockDisplay` contiene dos constructores. Lo que esos dos constructores proporcionan son formas alternativas de inicializar un objeto `ClockDisplay`. Si se utiliza el constructor que no tiene parámetros, entonces la hora inicial mostrada en el reloj será 00:00. Por otro lado, si queremos tener una hora inicial distinta, podemos configurarla utilizando el segundo constructor. Es común que las definiciones de clases contengan versiones alternativas de los constructores o de los métodos que proporcionan diversas formas de llevar a cabo una tarea concreta y que se diferencian entre sí por su conjuntos de parámetros. Esto se conoce con el nombre de *sobrecarga* de un constructor o de un método.

Ejercicio 3.28 Examine el segundo constructor en el código fuente de `ClockDisplay`. Explique qué hace y cómo lo hace.

Ejercicio 3.29 Identifique las similitudes y diferencias entre los dos constructores. Por ejemplo, ¿por qué no hay una llamada a `updateDisplay` en el segundo constructor?

3.11 Llamadas a métodos

3.11.1 Llamadas a métodos internos

La última línea del primer constructor `ClockDisplay` consta de la instrucción:

```
updateDisplay();
```

Concepto:

Los métodos pueden invocar a otros métodos de la misma clase como parte de su implementación. Esto se denomina **llamada a método interno**.

Esta instrucción es una *llamada a método*. Como hemos visto anteriormente, la clase `ClockDisplay` tiene un método con la siguiente signatura:

```
private void updateDisplay()
```

Lo que hace esa llamada a método es invocar precisamente este método. Puesto que este método se encuentra dentro la misma clase que la propia llamada al método, decimos que es una *llamada a un método interno*. Las llamadas a métodos internos tienen la sintaxis

nombreMetodo (lista-parámetros)

En nuestro ejemplo, el método no tiene ningún parámetro, por lo que la lista de parámetros está vacía. Esto se indica mediante la pareja de paréntesis sin nada en su interior.

Cuando se encuentra una llamada a método, se ejecuta el método correspondiente, y luego la ejecución vuelve a la llamada a método y continúa con la siguiente instrucción situada después de la misma. Para que la signatura de un método se corresponda con la llamada a método, tanto el nombre como la lista de parámetros deben corresponderse. Aquí, ambas listas de

parámetros están vacías, así que se corresponden. Esta necesidad de ajustarse tanto al nombre del método como a la lista de parámetros es importante, porque puede haber más de un método con el mismo nombre dentro de una clase, en caso de que ese método esté sobrecargado.

En nuestro ejemplo, el propósito de esta llamada a método es actualizar la cadena de visualización. Después de haber creado las dos pantallas numéricas, se configura la cadena de visualización para mostrar la hora indicada por los dos objetos de pantalla numérica. Más adelante explicaremos la implementación del método `updateDisplay`.

3.11.2 Llamadas a métodos externos

Examinemos ahora el siguiente método: `timeTick`. Su definición es:

```
public void timeTick()
{
    minutes.increment();
    if(minutes.getValue() == 0) {    // ¡ha vuelto a cero!
        hours.increment();
    }
    updateDisplay();
}
```

Si esta pantalla estuviera conectada a un reloj real, este método sería invocado una vez cada 60 segundos por el temporizador electrónico del reloj. Por el momento, nosotros nos limitamos a invocarlo nosotros mismos, con el fin de probar la pantalla.

Cuando se invoca el método `timeTick`, primero ejecuta la instrucción

```
minutes.increment();
```

Concepto:

Los métodos pueden invocar métodos de otros objetos, utilizando la notación con punto. Esto se denomina **llamada a método externo**.

Esta instrucción llama al método `increment` del objeto `minutes`. Por tanto, cuando se invoca uno de los métodos del objeto `ClockDisplay`, este a su vez llama a un método de otro objeto para llevar a cabo parte de la tarea. Las llamadas a métodos de otro objeto se denominan *llamadas a métodos externos*. La sintaxis de una llamada a método externo es

objeto . nombreMetodo (lista-parámetros)

Esta sintaxis se conoce como *notación con punto*. Está compuesta por un nombre de objeto, un punto, el nombre del método y los parámetros para la llamada. Es muy importante darse cuenta de que lo que utilizamos aquí es el nombre de un *objeto* y no el nombre de una clase. Empleamos el nombre `minutes` en lugar de `NumberDisplay`.

El método `timeTick` tiene a continuación una instrucción `if` para comprobar si también es necesario incrementar las horas. Como parte de la condición de la instrucción `if`, invoca otro método del objeto `minutes`: `getValue`. Este método devuelve el valor actual de los minutos. Si dicho valor es cero, entonces sabremos que la pantalla acaba de volver a cero y que debemos incrementar también las horas. Esto es exactamente lo que hace el código.

Si el valor de los minutos no es cero, entonces habremos terminado. No necesitamos cambiar las horas en ese caso. Por tanto, la instrucción `if` no necesita una parte *else*.

Con esto deberíamos poder comprender los tres métodos restantes de la clase `ClockDisplay` (véase el Código 3.4). El método `setTime` admite dos parámetros (la hora y los minutos) y configura el reloj con los valores especificados. Examinando el cuerpo del método podemos ver que hace esto invocando los métodos `setValue` de ambas pantallas numéricas (la de

las horas y la de los minutos). A continuación, llama a `updateDisplay` para actualizar correspondientemente la pantalla de visualización igual que hace el constructor.

El método `getTime` es trivial, simplemente devuelve la cadena de visualización actual. Puesto que siempre mantenemos la cadena de visualización actualizada, no hace falta hacer nada más.

Por último, el método `updateDisplay` es responsable de actualizar la cadena de visualización de modo que refleje correctamente la hora actual, representada por los dos objetos de pantalla numérica. Se invoca cada vez que cambia en el instante actual indicado por el reloj. Funciona llamando a los métodos `getDisplayValue` de cada uno de los objetos `NumberDisplay`. Estos métodos devuelven el valor de cada una de la pantallas numéricas. A continuación, utiliza la concatenación de cadenas para concatenar estos dos valores con un signo de dos puntos entre ellos, con el fin de formar una sola cadena.

Ejercicio 3.30 Dada una variable

```
Printer p1;
```

que almacena actualmente una referencia a un objeto impresora (printer) y dados dos métodos dentro de la clase `Printer` con las cabeceras

```
public void print(String filename, boolean doubleSided)
public int getStatus(int delay)
```

escriba dos llamadas posibles a cada uno de estos métodos.

3.11.3 Resumen de la pantalla de reloj

Merece la pena detenernos un momento a examinar la forma en que se utiliza la abstracción en este ejemplo con el fin de dividir el problema en partes más pequeñas. Examinando el código fuente de la clase `ClockDisplay`, podrá observar que nos limitamos a crear un objeto `NumberDisplay` sin que nos interese especialmente el aspecto interno de dicho objeto. Después podemos invocar métodos (`increment`, `getValue`) de dicho objeto para controlarlo. En este nivel, nos limitamos a asumir que el método de incremento se encargará de incrementar correctamente el valor de la pantalla, sin preocuparnos de cómo lo hace.

En los proyectos reales, a menudo, distintas personas escriben esas diferentes clases. Es posible que ya se haya dado cuenta de que lo único en lo que esas distintas personas tienen que ponerse de acuerdo es en qué signaturas de métodos tiene que tener cada clase y qué tienen que hacer esos métodos. Después, una persona podrá concentrarse en implementar esos métodos, mientras que otra puede limitarse a utilizarlos.

El conjunto de métodos que un objeto pone a disposición de otros objetos se denomina *interfaz*. Veremos las interfaces con mucho más detalle más adelante en el libro.

Ejercicio 3.31 *Ejercicio avanzado.* Cambie el reloj de uno de 24 horas a otro de 12 horas. Tenga cuidado: esto no es tan sencillo como puede parecer en principio. En un reloj de 12 horas, las horas después de medianoche y después del mediodía no se muestran como 00:30, sino como 12:30. Por tanto, la pantalla correspondiente a los minutos muestra valores comprendidos entre 0 y 59, mientras que la pantalla de las horas muestra valores comprendidos entre 1 y 12.

Ejercicio 3.32 Existen (al menos) dos formas en las que se puede implementar un reloj de 12 horas. Una posibilidad consiste simplemente en almacenar valores horarios entre 1 y 12. Pero, por otro lado, podríamos simplemente dejar que el reloj funcione internamente como un reloj de 24 horas, y cambiar la cadena de visualización de la pantalla del reloj para que muestre `4:23` o `4.23pm` cuando el valor interno sea `16:23`. Implemente ambas versiones. ¿Qué opción es más fácil? ¿Cuál de ellas es mejor? ¿Por qué?

3.12 Otro ejemplo de interacción entre objetos

Ahora vamos a examinar los mismos conceptos con un ejemplo distinto y utilizando diferentes herramientas. Seguimos interesados en tratar de comprender cómo los objetos crean otros objetos y cómo invocan los métodos de otros objetos. En la primera parte de este capítulo, hemos utilizado la técnica más fundamental para analizar cualquier programa dado: la lectura del código. La capacidad de leer y comprender el código fuente es una de las habilidades más esenciales para un desarrollador de software, y tendremos que aplicarla en todo proyecto en el que trabajemos. Sin embargo, en ocasiones resulta ventajoso utilizar herramientas adicionales para entender mejor cómo se ejecuta un programa. Una de esas herramientas, a la que ahora vamos a echar un vistazo es el *depurador*.

Concepto:

Un **depurador** es una herramienta software que ayuda a examinar cómo se ejecuta una aplicación. Se puede utilizar para localizar errores.

Un depurador es un programa que permite a los programadores ejecutar una aplicación paso a paso. Normalmente, proporciona funciones para iniciar y detener un programa en puntos seleccionados del código fuente, así como para examinar los valores de las variables.

El nombre *debugger* En inglés, los errores en programas informáticos se los denomina coloquialmente *bugs*. Por ello, a los programas depuradores que ayudan a eliminar esos errores se les conoce con el nombre de *debuggers*. No está del todo claro de dónde proviene el término *bug*, que en inglés significa "insecto". Hay una famosa anécdota de lo que se conoce como "el primer error informático", que fue debido a un insecto real (de hecho, a una polilla). Ese insecto fue encontrado, en 1945, dentro de la computadora Mark II por Grace Murray Hopper, una de las primeras personas que trabajó en el campo de la informática. Todavía se conserva en el Museo Nacional de Historia Americana del Instituto Smithsoniano un libro de registro en el que aparece una entrada con esta polilla pegada con cinta adhesiva al libro y con la anotación "Primer caso real de localización de un insecto (*bug*)". Sin embargo, tal como está redactada esa anotación, se sugiere que el término *bug* ya había estado utilizándose antes de que este insecto real causara problemas en el Mark II.

Si desea conocer más detalles, haga una búsqueda en la Web de la frase "*first computer bug*"-podrá encontrar incluso imágenes de esa polilla.

Los depuradores varían mucho en cuanto a complejidad. Los utilizados por desarrolladores profesionales tienen una gran cantidad de funciones que resultan útiles para hacer exámenes sofisticados de múltiples facetas de una aplicación. BlueJ tiene un depurador incorporado que es mucho más simple. Podemos utilizarlo para detener nuestro programa, para ejecutar el código línea por línea y para examinar los valores de las variables. Sin embargo, a pesar de la aparente falta de sofisticación de este depurador, es más que suficiente para obtener una gran cantidad de información.

Antes de comenzar a experimentar con el depurador, echaremos un vistazo al ejemplo que vamos a emplear para ilustrar la depuración: una simulación de un sistema de correo electrónico.

3.12.1 El ejemplo del sistema de correo

Comenzaremos investigando la funcionalidad del proyecto *mail-system*. En este instante, no necesitamos leer el código fuente, sino simplemente ejecutar el proyecto existente para tratar de comprender lo que hace.

Ejercicio 3.33 Abra el proyecto *mail-system*, que puede encontrar dentro del material de soporte del libro. La idea de este proyecto consiste en simular la actuación de una serie de usuarios que intercambian correos electrónicos. Un usuario emplea un cliente de correo para enviar mensajes de correo a un servidor, con el fin de que sean entregados al cliente de correo de otro usuario. Primero, cree un objeto `MailServer`. A continuación, cree un objeto `MailClient` para uno de los usuarios. Cuando cree el cliente necesitará suministrar una instancia `MailServer` como parámetro; utilice la que acaba de crear. También tendrá que especificar un nombre de usuario para el cliente de correo. Ahora cree un segundo cliente `MailClient` de forma similar, pero con un nombre de usuario diferente.

Experimente con los objetos `MailClient`. Pueden utilizarse para enviar mensajes de correo de un cliente de correo a otro (mediante el método `sendMailItem`), así como para recibir mensajes (usando los métodos `getNextMailItem` o `printNextMailItem`).

Examinando el proyecto de sistema de correo, vemos que:

- Tiene tres clases: `MailServer`, `MailClient` y `MailItem`.
- Es necesario crear un objeto servidor de correo que sea utilizado por todos los clientes de correo. Ese objeto se encargará del intercambio de mensajes.
- Se pueden crear varios objetos cliente de correo. Cada cliente de correo tendrá un nombre de usuario asociado.
- Se pueden enviar mensajes de correo de un cliente de correo a otro mediante un método contenido en la clase correspondiente al cliente de correo.
- Los mensajes de correo pueden ser recibidos por los clientes desde el servidor de uno en uno, utilizando un método contenido en el cliente de correo.
- La clase `MailItem`, que representa los mensajes de correo, nunca es instanciada de forma explícita por el usuario. Por el contrario, es utilizada internamente en el servidor y en los clientes de correo para crear, almacenar e intercambiar mensajes.

Ejercicio 3.34 Dibuje un diagrama de objetos de la situación existente después de crear un servidor de correo y tres clientes de correo. Los diagramas de objetos se han explicado en la Sección 3.6

Las tres clases tienen diferentes grados de complejidad. `MailItem` es bastante trivial. Vamos a analizar únicamente un pequeño detalle y dejaremos el resto para que el lector lo investigue. Teniendo en cuenta lo que hemos visto hasta ahora en el libro, `MailServer` es bastante compleja; hace uso de conceptos que explicaremos mucho más adelante en el libro. Por ello, no vamos a analizar ahora esta clase en detalle. En vez de eso, nos limitaremos a confiar en que esa clase hace correctamente su tarea, lo que constituye otro ejemplo de la forma en que se utiliza la abstracción para ocultar aquellos detalles de los que no tenemos que ser conscientes.

La clase `MailClient` es la más interesante, por lo que la examinaremos con cierto detalle.

3.12.2 La palabra clave `this`

La única sección que vamos a comentar de la clase `MailItem` es el constructor, que utiliza una estructura Java con la que hasta el momento no nos habíamos encontrado. El código fuente se muestra en el Código 3.5.

Código 3.5
Campos y constructor de la clase `MailItem`.

```
public class MailItem
{
    // El emisor del correo.
    private String from;
    // El destinatario del correo.
    private String to;
    // El texto del mensaje.
    private String message;

    /**
     * Crea un elemento de correo del emisor al destinatario
     * que contiene el mensaje deseado.
     * @param from    El emisor de este elemento.
     * @param to      El destinatario de este elemento.
     * @param message El texto del mensaje que hay que enviar.
     */
    public MailItem(String from, String to, String message)
    {
        this.from = from;
        this.to = to;
        this.message = message;
    }

    Se omiten los métodos.
}
```

La nueva funcionalidad Java en este fragmento de código es el uso de la palabra clave `this`:

```
this.from = from;
```

Toda la línea es una instrucción de asignación, que asigna el valor del lado derecho (`from`) a la variable del lado izquierdo (`this.from`).

La razón de emplear esta estructura es que tenemos una situación que se conoce con el nombre de *sobrecarga de nombres* –es la situación en que se utiliza el mismo nombre para dos entidades diferentes. La clase contiene tres campos, denominados `from`, `to` y `message`. El constructor tiene tres parámetros, también denominados `from`, `to` y `message`.

De modo que, mientras que estamos ejecutando el constructor, ¿cuántas variables existen? La respuesta es seis: tres campos y tres parámetros. Es importante entender que los campos y los parámetros son variables diferentes, que existen independientemente unas de otras, aun cuando compartan nombres similares. El que un parámetro y un campo compartan un nombre no es ningún problema en Java.

Pero lo que sí es un problema, sin embargo, es cómo referenciar las seis variables para poder distinguir entre los dos conjuntos de valores. Si simplemente usamos el nombre de variable "`from`" en el constructor (por ejemplo, en una instrucción `System.out.println(from)`), ¿qué variable se utilizaría, el parámetro o el campo?

La especificación del lenguaje Java permite responder a esta pregunta. Especifica que se utilice siempre la definición que tenga su origen en el bloque circundante más próximo. Puesto que el parámetro `from` está definido en el constructor y el campo `from` está definido en la clase, se utilizará el parámetro. Su definición está "más próxima" a la instrucción que la utiliza.

Ahora, todo lo que necesitamos es un mecanismo que nos permita acceder a un campo cuando haya una variable definida más próxima con el mismo nombre. Esto es precisamente para lo que se utiliza la palabra clave `this`. La expresión `this` hace referencia al objeto actual. Escribir `this.from` hace referencia al campo `from` del objeto actual. Por tanto, esta estructura nos da un medio de referirnos al campo, en lugar de al parámetro que tiene el mismo nombre. Con esto, podemos volver a leer la instrucción de asignación:

```
this.from = from;
```

Esta instrucción, como ahora vemos, tiene el siguiente efecto:

campo denominado `from` = *parámetro denominado* `from;`

En otras palabras, asigna el valor del parámetro al campo que tiene el mismo nombre. Esto es, por supuesto, exactamente lo que necesitamos para inicializar el objeto adecuadamente.

Nos queda por responder a una última cuestión: ¿por qué estamos haciendo todo esto? Podríamos evitar fácilmente todo el problema simplemente proporcionando a los campos y a los parámetros nombres distintos. La razón por la que no lo hacemos así es la legibilidad del código fuente.

En ocasiones, hay un nombre que describe perfectamente el uso de la variable –encaja tan bien que no queremos inventar un nombre distinto. Queremos utilizarlo para el parámetro, de modo que sirva como pista para el llamante, que indique lo que hace falta pasar; y también lo queremos usar para el campo, donde resulta útil como recordatorio para el implementador de la clase, ya que indica para qué se va a emplear el campo. Si un cierto nombre describe perfectamente el uso, resulta razonable emplearlo en ambos casos y aceptar la complicación de utilizar la palabra clave `this` dentro de la asignación para resolver el conflicto de nombres.

3.13 Uso de un depurador

La clase más interesante en el ejemplo del sistema de correo es el cliente de correo. Vamos a estudiarla ahora en más detalle utilizando un depurador. El cliente de correo tiene tres métodos: `getNextMailItem`, `printNextMailItem` y `sendMailItem`. Analizaremos primero el método `printNextMailItem`.

Antes de comenzar con el depurador, configure un escenario que podamos utilizar para nuestras investigaciones (Ejercicio 3.35).

Ejercicio 3.35 Configure un escenario para nuestras investigaciones. Cree un servidor de correo, a continuación cree dos clientes de correo para los usuarios "`Sophie`" y "`Juan`" (debe denominar a las instancias `sophie` y `juan` también para poder distinguirlas mejor en el banco de objetos). Después utilice el método `sendMailItem` de Sophie para enviar un mensaje a Juan. No lea el mensaje todavía.

Después de configurado el escenario como se indica en el Ejercicio 3.35, tenemos una situación en la que hay almacenado un elemento de correo en el servidor para Juan, esperando a ser extraído. Hemos visto que el método `printNextMailItem` extrae este elemento de correo y lo imprime en el terminal. Lo que queremos ahora es ver exactamente cómo funciona esto.

3.13.1 Establecimiento de puntos de interrupción

Para empezar nuestra investigación, vamos a configurar un punto de interrupción (Ejercicio 3.36). Un punto de interrupción es un indicador asociado a una línea de código fuente, que hará que se detenga la ejecución de un método en el momento en que se alcance ese punto. Se representa en el editor de BlueJ mediante un pequeño símbolo de *stop* (Figura 3.5).

Podemos establecer un punto de interrupción abriendo el editor de BlueJ, seleccionando la línea apropiada (en nuestro caso, la primera línea del método `printNextMailItem`) y luego seleccionando Set/Clear Breakpoint (Configurar/Eliminar punto de interrupción) en el menú *Tools* (herramientas) del editor. Alternativamente, podemos también hacer clic en el área situada junto a la línea de código en la que aparece el símbolo del punto de interrupción, con el fin de establecer o eliminar puntos de interrupción. Observe que para hacer esto es necesario compilar la clase.

Ejercicio 3.36 Abra el editor para la clase `MailClient` y establezca un punto de interrupción en la primera línea del método `printNextMailItem`, como se muestra en la Figura 3.5.

Una vez establecido el punto de interrupción, invoque el método `printNextMailItem` en el cliente de correo de Juan. Se abrirán la ventana del editor para la clase `MailClient` y una ventana del depurador (Figura 3.6).

En la parte inferior de la ventana del depurador hay disponibles algunos botones de control. Pueden utilizarse para continuar o interrumpir la ejecución del programa. (Para ver una explicación más detallada de los controles del depurador, consulte el Apéndice F.)

Figura 3.5
Un punto de interrupción en el editor de BlueJ.

MailClient

Compile | Undo | Cut | Copy | Paste | Find... | Close | Source Code

```
/**
 * Print the next mail item (if any) for this user to the text
 * terminal.
 */
public void printNextMailItem()
{
    MailItem item = server.getNextMailItem(user);
    if(item == null) {
        System.out.println("No new mail.");
    }
    else {
        item.print();
    }
}

/**
 * Send the given message to the given recipient via
 * the attached mail server.
```

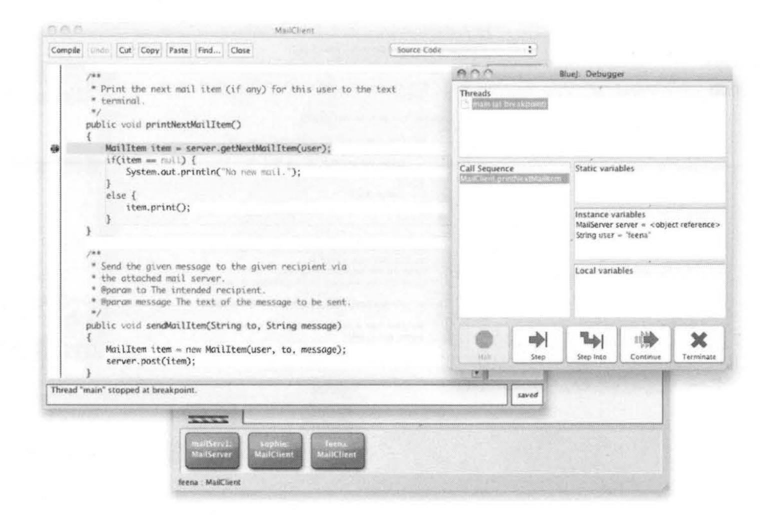


Figura 3.6
La ventana del depurador, la ejecución se ha detenido en un punto de interrupción.

En la parte derecha de la ventana del depurador hay tres áreas para la visualización de variables, denominadas *static variables*, *instance variables* y *local variables*, que se utilizan respectivamente para visualizar las variables estáticas, de instancia y locales. Por el momento, vamos a ignorar el área correspondiente a las variables estáticas. Hablaremos de estas variables más adelante, y además esta clase no tiene ninguna.

Vemos que este objeto tiene dos variables de instancia (o campos), `server` y `user`, y podemos ver también los valores actuales. La variable `user` almacena la cadena `"Juan"` y la variable `server` almacena una referencia a otro objeto. La referencia a objeto es lo que hemos dibujado mediante una flecha en los diagramas de objetos.

Observe que todavía no hay ninguna variable local. Esto es porque la ejecución se ha detenido *antes* de ejecutar la línea en la que se ha definido el punto de interrupción. Puesto que la línea con el punto de interrupción contiene la única variable local y dicha línea todavía no ha sido ejecutada, no existe ninguna variable local en este momento.

El depurador no solo nos permite interrumpir la ejecución del programa e inspeccionar las variables, sino que también nos permite avanzar en la ejecución lentamente.

3.13.2 Ejecución paso a paso

Al estar parados en un punto de interrupción, si hacemos clic en el botón *Step* (paso) se ejecuta una única línea de código y luego la ejecución vuelve a detenerse.

Ejercicio 3.37 Avance una línea en la ejecución del método `printNextMailItem` haciendo clic en el botón *Step*.

El resultado de ejecutar la primera línea del método `printNextMailItem` se muestra en la Figura 3.7. Podemos ver que la ejecución ha avanzado una línea (una pequeña flecha negra junto a la línea del código fuente indica cuál es la posición actual), y la lista de variables locales en la ventana del depurador indica que se ha creado una variable local y que se la ha asignado un objeto.

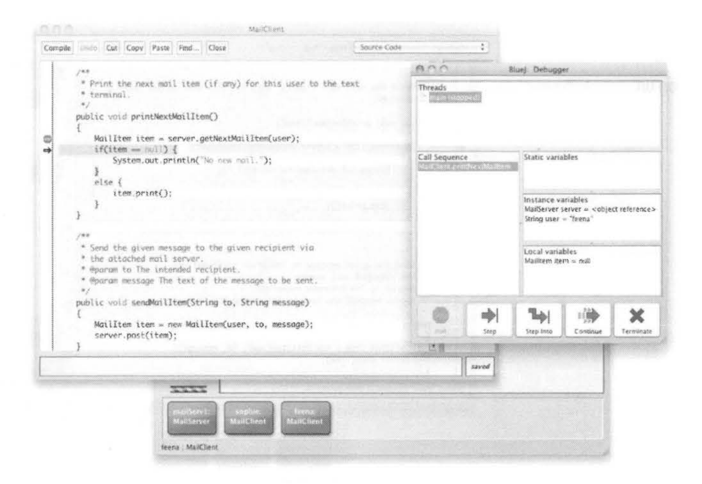


Figura 3.7
Otra vez parados, después de hacer avanzar un único paso la ejecución.

Ejercicio 3.38 Prediga qué línea se marcará como siguiente línea a ejecutar después del siguiente paso. Después ejecute otro paso y compruebe su predicción. ¿Acertó? Explique lo que ha sucedido y por qué.

Ahora podemos utilizar repetidamente el botón *Step* para ir paso a paso hasta el final del método. Esto nos permite ver la ruta que toma la ejecución del programa. Esto resulta especialmente interesante en las instrucciones condicionales: podemos ver claramente qué rama de una instrucción `if` se ejecuta y emplear esa información para comprobar si el flujo de ejecución está de acuerdo con lo que nosotros esperábamos.

Ejercicio 3.39 Vuelva a invocar el mismo método (`printNextMailItem`). Ejecute el método paso a paso como antes. ¿Qué es lo que observa? Explique por qué sucede.

3.13.3 Entrada en un método durante la ejecución paso a paso

Al ejecutar paso a paso el método `printNextMailItem`, hemos visto dos llamadas a métodos de objetos de nuestras propias clases. La línea

```
MailItem item = server.getNextMailItem(user);
```

incluye una llamada al método `getNextMailItem` del objeto `server`. Comprobando las declaraciones de las variables de instancia, podemos ver que el objeto `server` está declarado como de clase `MailServer`.

La línea

```
item.print();
```

llama al método `print` del objeto `item`. Podemos ver en la primera línea del método `printNextMailItem` que `item` está declarado como de clase `MailItem`.

Utilizando el comando *Step* del depurador, hemos empleado la abstracción: hemos estado contemplando el método `print` de la clase `item` como una única instrucción, y hemos podido observar que su efecto es el de imprimir los detalles (emisor, destinatario y mensaje) del elemento de correo.

Si estamos interesados en obtener más detalles, podemos examinar más a fondo el proceso y ver ejecutarse paso a paso el método `print`. Esto se hace empleando el comando *Step Into* en el depurador en lugar del comando *Step*. *Step Into* entrará dentro del método que se está invocando y se detendrá en la primera línea de dicho método.

Ejercicio 3.40 Configure el mismo escenario de prueba que antes. Es decir, envíe un mensaje de Sophie a Juan. Luego invoque de nuevo el método `printNextMailItem` del cliente de correo de Juan. Realice una ejecución paso a paso como antes, pero esta vez, cuando llegue a la línea

```
item.print();
```

utilice el comando *Step Into* en lugar del comando *Step*. Asegúrese de poder ver la ventana de terminal de texto mientas avanza paso a paso. ¿Qué ha observado? Explique lo que vea.

3.14 Un nuevo análisis de las llamadas a métodos

En los experimentos de la Sección 3.13, hemos visto otro ejemplo de interacción entre objetos similar a la que vimos antes: objetos que llaman a métodos de otros objetos. En el método `printNextMailItem`, el objeto `MailClient` hacía una llamada a un objeto `MailServer` con el fin de extraer el siguiente elemento de correo. Este método (`getNextMailItem`) devolvía un valor, un objeto de tipo `MailItem`. Después había una llamada al método `print` del elemento de correo. Utilizando la abstracción, podemos ver el método `print` como un único comando, o bien, si nos interesa conocer más detalles, podemos pasar a un nivel inferior de abstracción y examinar el interior del método `print`.

De forma similar, podemos utilizar el depurador para observar un objeto mientras crea otro. El método `sendMessage` de la clase `MailClient` es un buen ejemplo. En este método, se crea un objeto `MailItem` en la primera línea de código.

```
MailItem item = new MailItem(user, to, message);
```

La idea aquí es que el elemento de correo se emplea para encapsular un mensaje de correo. El elemento de correo contiene información acerca del emisor, del destinatario y del propio mensaje. Cuando se envía un mensaje, un cliente de correo crea un elemento de correo con toda esta información y luego lo almacena en el servidor de correo. Allí, puede ser extraído posteriormente por el cliente de correo del destinatario.

En la línea de código anterior, vemos que se utiliza la palabra clave `new` para crear el nuevo objeto, y vemos cómo se pasan los parámetros al constructor. (Recuerde: construir un objeto hace dos cosas, crear un objeto y ejecutar el constructor.) Invocar el constructor funciona de forma muy similar a la invocación de métodos. Esto se puede observar utilizando el comando *Step Into* en la línea en la que el objeto está siendo construido.

Ejercicio 3.41 Establezca un punto de interrupción en la primera línea del método `sendMailItem` en la clase `MailClient`. A continuación invoque este método. Utilice la función *Step Into* para entrar en el constructor del elemento de correo. En la pantalla del depurador para el objeto `MailItem`, podrá ver las variables de instancia y las variables locales que tienen los mismos nombres, como se explica en la Sección 3.12.2. Haga una ejecución paso a paso para ver cómo se inicializan las variables de instancia.

Ejercicio 3.42 Utilice una combinación de lectura de código, ejecución de métodos, puntos de interrupción y ejecución paso a paso para familiarizarse con las clases `MailItem` y `MailClient`. Observe que todavía no hemos dado las suficientes explicaciones como para poder comprender la implementación de la clase `MailServer`, por lo que por el momento puede ignorarla. Por supuesto, puede examinarla si tiene ganas de experimentar, pero no se sorprenda si la encuentra ligeramente compleja... Explique por escrito cómo interactúan las clases `MailClient` y `MailItem`. Dibuje los diagramas de objetos como parte de sus explicaciones.

3.15 Resumen

En este capítulo hemos explicado cómo se puede dividir un problema en subproblemas. Podemos tratar de identificar subcomponentes en aquellos objetos que deseemos modelar y podemos implementar los subcomponentes mediante clases independientes. Hacer esto así ayuda a reducir la complejidad de implementación de las aplicaciones de gran tamaño, porque nos permite implementar, probar y mantener las clases individuales por separado.

Hemos visto cómo esto nos proporciona estructuras de objetos que funcionan conjuntamente para resolver una tarea común. Los objetos pueden crear otros objetos y pueden invocar los métodos de otros objetos. Entender estas interacciones entre los objetos es esencial para planificar, implementar y depurar aplicaciones.

Podemos emplear diagramas hechos con lápiz y papel, lectura de código y depuradores para investigar cómo se ejecuta una aplicación o para localizar errores.

Términos introducidos en el capítulo

abstracción, modularización, divide y vencerás, diagrama de clases, diagrama de objetos, referencia a objeto, sobrecarga, llamada a método interno, llamada a método externo, notación con punto, depurador, punto de interrupción

Resumen de conceptos

- **abstracción** La abstracción es la capacidad de ignorar los detalles de las distintas partes, para centrar la atención en un nivel superior de un problema.
- **modularización** La modularización es el proceso de dividir un todo en partes bien definidas que puedan construirse y examinarse por separado y que interactúen de formas bien definidas.
- **clases definen tipos** Un nombre de clase puede utilizarse como tipo para una variable. Las variables que tienen una clase como tipo pueden almacenar objetos de dicha clase.
- **diagrama de clases** El diagrama de clases muestra las clases de una aplicación y las relaciones entre ellas. Proporciona información acerca del código fuente y presenta una vista estática de un programa.
- **diagrama de objetos** El diagrama de objetos muestra los objetos y sus relaciones en un instante determinado durante la ejecución de una aplicación. Proporciona información acerca de los objetos en tiempo de ejecución y presenta una vista dinámica de un programa.
- **referencias a objeto** Las variables con un tipo de objeto almacenan referencias a objetos.

- **tipos primitivos** Los tipos primitivos en Java son los tipos que no son de objeto. Los tipos primitivos más comunes son `int`, `boolean`, `char`, `double` y `long`. Los tipos primitivos no disponen de métodos.
- **creación de objetos** Los objetos pueden crear otros objetos, utilizando el operador `new`.
- **sobrecarga** Una clase puede contener más de un constructor o más de un método con el mismo nombre, siempre que cada uno tenga un conjunto diferente de tipos de parámetros.
- **llamada a método interno** Los métodos pueden invocar a otros métodos de la misma clase como parte de su implementación. Esto se denomina llamada a método interno.
- **llamada a método externo** Los métodos pueden invocar métodos de otros objetos, utilizando la notación con punto. Esto se denomina llamada a método externo.
- **depurador** Un depurador es una herramienta software que ayuda a examinar cómo se ejecuta una aplicación. Se puede utilizar para localizar errores.

Ejercicio 3.43 Utilice el depurador para investigar el proyecto *clock-display*. Establezca puntos de interrupción en el constructor `ClockDisplay` y en cada uno de los métodos y luego ejecútelos paso a paso. ¿Se comporta el programa como esperaba? ¿Le ha proporcionado esto más conocimientos? En caso afirmativo, ¿cuáles son?

Ejercicio 3.44 Utilice el depurador para investigar el método `insertMoney` del proyecto *better-ticket-machine* del Capítulo 2. Realice pruebas que hagan que se ejecuten ambas ramas de la instrucción `if`.

Ejercicio 3.45 *Ejercicio avanzado.* Añada una línea de asunto del correo electrónico a los elementos de correo del proyecto *mail-system*. Asegúrese de que al imprimir los mensajes también se imprima la línea de asunto. Modifique el cliente de correo según sea necesario.

Ejercicio 3.46 Dada la siguiente clase (de la que solo se muestran aquí algunos fragmentos),

```
public class Screen
{
    public Screen(int xRes, int yRes)
    { ...
    }

    public int numberOfPixels()
    { ...
    }

    public void clear(boolean invert)
    { ...
    }
}
```

escriba algunas línea de código Java que permitan crear un objeto `Screen`, que representa a una pantalla. Después llame a su método `clear` si (y solo si) su número de píxeles es superior a dos millones. (No se preocupe ahora por la lógica del ejemplo, el objetivo es únicamente escribir algo que sea sintácticamente correcto; es decir, que se compile correctamente si lo escribimos.)

CAPÍTULO

4

Agrupación de objetos

Principales conceptos explicados en el capítulo:

- colecciones
- bucles
- iteradores
- matrices

Estructuras Java explicadas en este capítulo:

`ArrayList`, `Iterator`, bucle `while`, `null`, cast, objetos anónimos, matriz, bucle `for`, bucle for-each, ++

El principal objeto de este capítulo es presentar algunas de las formas en las que pueden agruparse objetos para formar colecciones. En particular, hablaremos de la clase `ArrayList` como ejemplo de colecciones de tamaño flexible y veremos cómo se usan los objetos matriz para colecciones de tamaño fijo. Estrechamente asociada con las colecciones se encuentra la necesidad de iterar a lo largo de los elementos que esas colecciones contienen. Con ese propósito, presentaremos tres nuevas estructuras de control: dos versiones del bucle `for` y el bucle `while`.

Este capítulo es muy largo y es además muy importante. No podrá convertirse en un buen programador sin entender perfectamente el contenido de este capítulo. Le hará falta dedicar más tiempo a su estudio que al de los capítulos precedentes, pero no se deje llevar por la tentación de apresurarse en su lectura; tómese su tiempo y estúdielo en profundidad.

4.1 Profundización en algunos conceptos del Capítulo 3

Además de presentar nuevo material sobre las colecciones y la iteración, también revisaremos dos de los temas clave vistos en el Capítulo 3: la *abstracción* y la *interacción entre objetos*. Allí vimos que la abstracción nos permite simplificar un problema, identificando componentes discretos que puedan contemplarse como un todo, en lugar de tener que preocuparnos por sus detalles. Veremos este principio en acción cuando comencemos a hacer uso de las *clases de librería* disponibles en Java. Aunque estas clases no son, estrictamente hablando, parte del lenguaje, algunas de ellas están íntimamente asociadas con la escritura de la mayor parte de los programas Java, por lo que a menudo se piensa en ellas como en una parte más del lenguaje. La mayoría de la gente que escribe programas Java comprueba constantemente

las librerías para ver si alguien ha escrito ya una clase que ellos puedan aprovechar. De esta forma, se ahorran una gran cantidad de esfuerzo, que puede emplearse mejor en trabajar en otras partes del programa. El mismo principio se aplica en la mayoría de los demás lenguajes de programación, que también suelen disponer de librerías de clases útiles. Por tanto, merece la pena familiarizarse con el contenido de la librería y saber cómo usar las clases más comunes. La potencia de la abstracción es que para poder usar una clase de manera efectiva, normalmente no nos hace falta conocer muchos detalles (¡de hecho puede que ninguno!) acerca de las interioridades de la clase.

Si utilizamos una clase de la librería, lo que haremos será escribir código que cree instancias de esa clase, después de lo cual nuestros objetos podrán interactuar con los objetos de la librería. Por tanto, la interacción entre objetos también desempeñará un importante papel en este capítulo.

A medida que progrese en la lectura, se encontrará con que los capítulos de este libro están volviendo continuamente sobre los temas que se han introducido en los capítulos anteriores, y profundizando en ellos. En el prefacio nos referíamos a estos cuando decíamos que el libro adopta un "enfoque iterativo". Una ventaja concreta de este enfoque es que ayuda a profundizar gradualmente en la compresión de los temas, a medida que se va progresando en la lectura del libro.

En este capítulo también profundizaremos en el tema de la abstracción, para ver que ese concepto no solo implica ocultar los detalles, sino que también implica ser capaces de detectar los patrones y características comunes que aparecen una y otra vez en los programas. Ser capaz de reconocer estos patrones nos permitirá a menudo reutilizar total o parcialmente, en una nueva situación, algún método o clase que hayamos escrito previamente. Esto se aplica, especialmente, al examinar las colecciones y la iteración.

4.2 La colección como abstracción

Una de las abstracciones que queremos explorar en este capítulo es la idea de *colección* –el concepto de agrupar cosas para poder referirnos a ellas y manejarlas de manera conjunta. Una colección puede ser: grande (todos los estudiantes de una universidad), pequeña (los cursos en que se ha matriculado un estudiante) o incluso puede estar vacía (¡los cuadros de Picasso que he adquirido a lo largo de mi vida!).

Si poseemos una colección de sellos, autógrafos, de posters de conciertos, de figuras decorativas, de discos o de cualquier otra cosa, entonces existirán cosas comunes que querremos hacer con la colección de vez en cuando, independientemente de qué sea lo que coleccionemos. Por ejemplo, es posible que queramos *añadir* elementos a la colección, pero puede también que queramos reducirla –por ejemplo, si tenemos duplicados o si queremos obtener dinero para hacer compras adicionales. También podríamos querer *ordenar* la colección de una cierta manera, por ejemplo por fecha de adquisición o por valor. Lo que estamos describiendo aquí son *operaciones* típicas que podemos efectuar sobre una colección.

Concepto:

Colección Un objeto colección puede almacenar un número arbitrario de otros objetos.

En un contexto de programación, la abstracción colección se convierte en una clase de algún tipo, y las operaciones serían los métodos de esa clase. Una colección (mi colección de música) sería una instancia de la clase. Además, los elementos almacenados en una instancia de colección serían, ellos mismos, objetos.

He aquí algunos otros ejemplos de colecciones que están más claramente relacionados con un contexto de programación:

- Los calendarios electrónicos almacenan notas de eventos relativas a citas, reuniones, cumpleaños, etc. Se van añadiendo nuevas notas a medida que se organizan eventos futuros y las notas antiguas se borran cuando los detalles de los eventos pasados dejan de ser necesarios.
- Las librerías guardan detalles acerca de los libros y revistas que poseen. El catálogo va cambiando a medida que se compran nuevos libros y que los antiguos se envían al almacén o se tiran.
- Las universidades mantienen registros de estudiantes. Cada año académico añade registros a la colección, mientras que los registros de aquellos que han abandonado la universidad se transfieren a una colección para archivado definitivo. En este contexto es habitual elaborar listados de subconjuntos de la colección: todos los estudiantes matriculados en una cierta asignatura o todos los estudiantes que tienen que graduarse este año, por ejemplo.

Resulta típico que el número de elementos almacenados en una colección varíe de vez en cuando. Hasta ahora, no hemos visto ninguna característica de Java que nos permita agrupar un número arbitrario de elementos. Quizá podríamos definir una clase con un montón de campos individuales, para abarcar un número fijo pero muy grande de elementos, pero los programas suelen necesitar una solución que sea más general que esta. Una solución adecuada sería aquella que no nos exigiera saber de antemano cuántos elementos vamos a agrupar, y que no nos obligara a fijar un límite superior para dicho número.

Por tanto, comenzaremos nuestra exploración de la librería Java buscando una clase que nos proporcione la forma más simple posible de agrupar objetos, mediante una lista secuencial desordenada de tamaño flexible: `ArrayList`. En las siguientes secciones, utilizaremos el ejemplo de la persona que quiere gestionar su colección de música particular para ilustrar cómo podemos agrupar un número arbitrario de objetos en un único objeto contenedor.

4.3 Un organizador para archivos de música

Vamos a escribir una clase que nos ayude a organizar nuestros archivos de música almacenados en una computadora. Nuestra clase no almacenará en realidad los detalles de los archivos; en su lugar, lo que hará será delegar esa responsabilidad en la clase estándar de librería `ArrayList`, que nos ahorrará un montón de trabajo. Pero entonces, ¿por qué necesitamos escribir una clase? Un punto importante que hay que tener en mente al tratar con las clases de librería es que estas no se han escrito para ningún escenario de aplicación completo, son clases de propósito general. Una instancia de `ArrayList` podría almacenar objetos que fueran registros de estudiantes, mientras que otra podría emplearse para almacenar recordatorios sobre eventos. Esto significa que quienes proporcionan las operaciones específicas de cada escenario son las clases que escribamos para utilizar las clases de librería; por ejemplo, será la clase que escribamos la que se encargue de reflejar el hecho de que estamos tratando con archivos de música o reproduciendo un archivo almacenado en la colección.

En aras de la simplicidad, la primera versión de este proyecto trabajará simplemente con los nombres de archivo de las pistas de música individuales. No habrá detalles independientes

relativos al título, al artista, a la duración, etc. Esto significa que simplemente pediremos a la instancia `ArrayList` que almacene objetos `String` que representen los nombres de archivo. El mantener un cierto grado de simplicidad en esta etapa nos ayudará a evitar que queden oscurecidos los conceptos clave que estamos tratando de ilustrar, que son la creación y utilización de un objeto colección. Posteriormente en el capítulo, añadiremos una mayor sofisticación, con el fin de obtener un organizador y reproductor de música más funcional.

Vamos a suponer que cada archivo de música representa una única pista de audio. Los archivos de ejemplo que proporcionamos con el proyecto tienen integrado en el nombre de archivo tanto el nombre del artista como el título de la canción, y emplearemos esa característica posteriormente. Pero por el momento, he aquí las operaciones básicas que incorporaremos en la versión inicial de nuestro organizador:

- Permite añadir canciones a la colección.
- No tiene ningún límite predeterminado en cuando al número de canciones que es posible almacenar, salvo por el propio límite de memoria de la máquina en la que se ejecute la aplicación.
- Nos dirá cuántas canciones hay en la colección.
- Permitirá enumerar todas las canciones.

Como veremos, la clase `ArrayList` hace que sea muy sencillo proporcionar esta funcionalidad a partir de nuestra propia clase.

Observe que no estamos siendo demasiado ambiciosos en esta primera versión. Estas características serán suficientes para ilustrar los fundamentos básicos de creación y uso de la clase `ArrayList`, mientras que en otras versiones posteriores iremos añadiendo características incrementalmente hasta disponer de un organizador más sofisticado. (Lo más importante, quizá, es que más adelante añadiremos la posibilidad de reproducir los archivos de música. Nuestra primera versión no será capaz de hacer eso por el momento.) Este enfoque modesto e incremental tiene muchas más posibilidades de tener éxito que el tratar de implementar todo de una vez.

Antes de analizar el código fuente necesario para hacer uso de una clase así, será útil explorar el comportamiento inicial del organizador de música.

Ejercicio 4.1 Abra el proyecto *music-organizer-v1* en BlueJ y cree un objeto `MusicOrganizer`. Almacene en él los nombres de unos cuantos archivos de audio, se trata simplemente de cadenas de caracteres. Como por el momento no vamos a reproducir los archivos, puede usar cualquier nombre que quiera aunque existe un muestrario de archivos de audio en la subcarpeta *audio* del proyecto que puede utilizar si lo desea.

Compruebe que el número de archivos devuelto por `numberOfFiles` se corresponde con el número que haya almacenado. Cuando utilice el método `listFile`, tendrá que utilizar como valor de parámetro 0 (cero) para imprimir el primer archivo, 1 (uno) para imprimir el segundo, etc. Explicaremos la razón de este sistema de numeración a su debido tiempo.

Ejercicio 4.2 ¿Qué sucede si se crea un nuevo objeto `MusicOrganizer` y luego se invoca `removeFile(0)` antes de haber añadido ningún archivo? ¿Se obtiene algún error? ¿Cabría esperar obtener algún error?

Ejercicio 4.3 Cree un objeto `MusicOrganizer` y añádale dos nombres de archivo. Invoque `listFile(0)` y `listFile(1)` para visualizar los dos archivos. Ahora llame a `removeFile(0)` y luego a `listFile(0)`. ¿Qué ha sucedido? ¿Es eso lo que esperaba? ¿Se le ocurre alguna explicación de lo que puede haber sucedido al eliminar el primer nombre de archivo de la colección?

4.4 Utilización de una clase de librería

El Código 4.1 muestra la definición completa de nuestra clase `MusicOrganizer`, que hace uso de la clase de librería `ArrayList`. Observe que las clases de librería no aparecen en el diagrama de clases de BlueJ.

Librerías de clases Una de las características de los lenguajes orientados a objetos que les dota de más potencia es que a menudo suelen estar acompañados por *librerías de clases*. Esta librerías suelen contener varios cientos o miles de clases distintas, que han demostrado ser útiles para los desarrolladores en un amplio rango de proyectos distintos. Java denomina a sus librerías *paquetes*. Las clases de librería se emplean exactamente de la misma forma que utilizaríamos nuestras clases. Las instancias se construyen utilizando `new` y las clases tienen campos, constructores y métodos.

Código 4.1
La clase `MusicOrganizer`.

```
import java.util.ArrayList;

/**
 * Una clase para almacenar detalles de archivos de audio.
 *
 * @author David J. Barnes y Michael Kölling
 * @version 2011.07.31
 */
public class MusicOrganizer
{
    // ArrayList para almacenar el nombre de los archivos de música.
    private ArrayList<String> files;

    /**
     * Crea un MusicOrganizer
     */
    public MusicOrganizer()
    {
        files = new ArrayList<String>();
    }


    /**
     * Añade un archivo a la colección.
     * @param filename El archivo que hay que añadir.
     */
```

Código 4.1
(continuación)
La clase
`MusicOrganizer`.

```
    public void addFile(String filename)
    {
        files.add(filename);
    }

    /**
     * Devuelve el número de archivos de la colección.
     * @return El número de archivos de la colección.
     */
    public int getNumberOfFiles()
    {
        return files.size();
    }

    /**
     * Muestra un archivo de la colección.
     * @param index El índice del archivo que hay que mostrar.
     */
    public void listFile(int index)
    {
        if(index >= 0 && index < files.size()) {
            String filename = files.get(index);
            System.out.println(filename);
        }
    }

    /**
     * Elimina un archivo de la colección.
     * @param index El índice del archivo que hay que eliminar.
     */
    public void removeFile(int index)
    {
        if(index >= 0 && index < files.size()) {
            files.remove(index);
        }
    }
```

4.4.1 Importación de una clase de librería

La primera línea del archivo de clase ilustra la forma en la que obtenemos acceso a una clase de librería en Java: mediante una *instrucción de importación* (`import`):

```
import java.util.ArrayList;
```

Esto hace que la clase `ArrayList` del paquete `java.util` esté disponible a la hora de definir nuestra clase. Las instrucciones de importación debe siempre colocarse antes de las

instrucciones de clases en un archivo. Una vez importado desde un paquete de esta manera un archivo de clase, podemos utilizar dicha clase como si fuera una de nuestras propias clases. Así, empleamos `ArrayList` al principio de la clase `MusicOrganizer` para definir un campo `files`:

```
private ArrayList<String> files;
```

Aquí podemos ver una nueva estructura sintáctica: la mención de `String` entre corchetes angulares, `<String>`. En cierto modo, ya aludimos anteriormente en la Sección 4.3 a la necesidad de hacer esto, cuando señalamos que `ArrayList` es una clase de colección de *propósito general* –es decir, que no está restringida en lo que respecta a los tipos de objeto que puede almacenar. Sin embargo, cuando creamos un objeto `ArrayList`, tenemos que ser específicos acerca del tipo de objetos que se almacenarán en esa instancia concreta. Podemos almacenar cualquier tipo que decidamos, pero es necesario designar dicho tipo al declarar una variable `ArrayList`. Las clases como `ArrayList`, que se parametrizan con un segundo tipo, se denominan *clases genéricas* (hablaremos de ellas con mayor detalle posteriormente).

Al utilizar colecciones, por tanto, siempre tenemos que especificar dos tipos: el tipo de la propia colección (en este caso, `ArrayList`) y el tipo de los elementos que pretendemos almacenar en esa colección (que aquí es `String`). Podemos leer la definición completa de tipo `ArrayList<String>` como *"una colección ArrayList de objetos de tipo String"*. En nuestra clase, usamos esa definición de tipo como tipo para nuestra variable `files`.

A estas alturas del libro, ya debería haberse acostumbrado a esperar que exista una estrecha conexión entre el cuerpo del constructor y los campos de la clase, porque el constructor es responsable de inicializar los campos de cada instancia. Así, de la misma forma que `ClockDisplay` creaba objetos `NumberDisplay` para sus dos campos, aquí podemos ver al constructor de `MusicOrganizer` creando un objeto de tipo `ArrayList<String>` y almacenándolo en el campo `files`.

4.4.2 Notación diamante

Observe que al crear la instancia `ArrayList` hemos especificado de nuevo el tipo completo, indicando el tipo de elemento entre corchetes angulares y poniendo a continuación unos paréntesis para la lista de parámetros (que está vacía):

```
files = new ArrayList<String>();
```

En todas las versiones de Java anteriores a la versión 7, era obligatorio emplear el formato completo del tipo genérico a la hora de crear una instancia, pero desde Java 7 el compilador Java puede inferir el tipo parametrizado del objeto que se está creando a partir del tipo de la variable a la que se está realizando la asignación. Esto nos permite utilizar la denominada *notación diamante* de la forma siguiente:

```
files = new ArrayList<>();
```

Utilizar esta forma no cambia el hecho de que el objeto que se está creando solo será capaz de almacenar objetos `String`; se trata simplemente de una característica que se ha introducido por comodidad para el programador.

Sin embargo, en este libro, seguiremos empleando la notación de estilo antiguo, primero por compatibilidad con la mayoría del código fuente Java con el que realmente vaya a encontrarse, y también, porque en el momento de escribir estas líneas, no todos los lectores del libro tendrán acceso a un compilador de Java 7.

4.4.3 Principales métodos de `ArrayList`

La clase `ArrayList` define un montón de métodos, pero nosotros, para dar soporte a la funcionalidad que nos hace falta, solo vamos a utilizar por el momento cuatro de ellos: `add`, `size`, `get` y `remove`.

Los dos primeros se ilustran, respectivamente, en los métodos `addFile` y `getNumberOfFiles`, que son relativamente sencillos. El método `add` de un `ArrayList` almacena un objeto en la lista y el método `size` nos dice cuántos elementos hay almacenados en la lista actualmente. En la Sección 4.7 veremos cómo funcionan los métodos `get` y `remove`, aunque probablemente pueda hacerse una idea simplemente leyendo el código de los métodos `listFile` y `removeFile`.

4.5 Estructuras de objetos con colecciones

Para entender cómo opera un objeto colección como `ArrayList`, resulta útil examinar un diagrama de objetos. La Figura 4.1 ilustra el aspecto que tendría un objeto `MusicOrganizer` con dos cadenas representativas de nombres de archivo almacenadas en él. Compare la Figura 4.1 con la Figura 4.2, en la que se ha almacenado un tercer nombre de archivo.

Existen al menos tres características de la clase `ArrayList` que debemos considerar:

- Es capaz de incrementar su capacidad interna según sea necesario: a medida que se añaden nuevos elementos, se limita a crear espacio para ellos.
- Mantiene su propio contador privado, el número de elementos que almacena en cada instante. Su método `size` devuelve el valor de ese contador.
- Mantiene el orden de los elementos que se inserten en la lista. El método `add` almacena cada nuevo elemento al final de la lista. Posteriormente, podemos extraerlos en el mismo orden.

Podemos observar que el objeto `MusicOrganizer` parece bastante simple, tiene un único campo que almacena un objeto de tipo `ArrayList<String>`. Todo el trabajo complicado se realiza dentro del objeto `ArrayList`. Esta es una de las grandes ventajas de utilizar clases de librería. Alguien ha invertido tiempo y esfuerzo para implementar algo útil, y nosotros tenemos acceso a esa funcionalidad de manera prácticamente gratuita sin más que utilizar esa clase.

Por el momento, no necesitamos preocuparnos por *cómo* es capaz un `ArrayList` de soportar esas características. Nos basta con ser capaces de apreciar lo útil que es esta capacidad. Recuerde: esta supresión de los detalles es uno de los beneficios que la abstracción nos proporciona; implica que podemos utilizar `ArrayList` para escribir cualquier número de clases distintas que necesiten almacenar un número arbitrario de objetos.

La segunda característica, el hecho de que el objeto `ArrayList` lleve la cuenta del número de objetos insertados, tiene consecuencias importantes para la forma en la que implementemos la clase `MusicOrganizer`. Aunque el organizador tiene un método `getNumberOfFiles`, no hemos definido ningún campo específico para almacenar esa información. En lugar de ello, el organizador *delega* la responsabilidad de llevar la cuenta del número de elementos en su objeto `ArrayList`. Esto significa que el organizador no duplica la información cuando la puede conseguir de alguna otra manera. Si un usuario solicita al organizador información

acerca del número de nombres de archivo que contiene, el organizador pasaría la pregunta al objeto `files` y luego devolvería la respuesta que este le proporcionara.

A menudo, como programadores, tendremos que esforzarnos para evitar duplicar la información o los comportamientos. La duplicación puede representar un desperdicio de esfuerzos y puede también provocar que aparezcan incoherencias, cuando dos cosas que deberían ser idénticas resultan no serlo, debido a algún tipo de error. En posteriores capítulos, profundizaremos en este tema de la duplicación de funcionalidad.

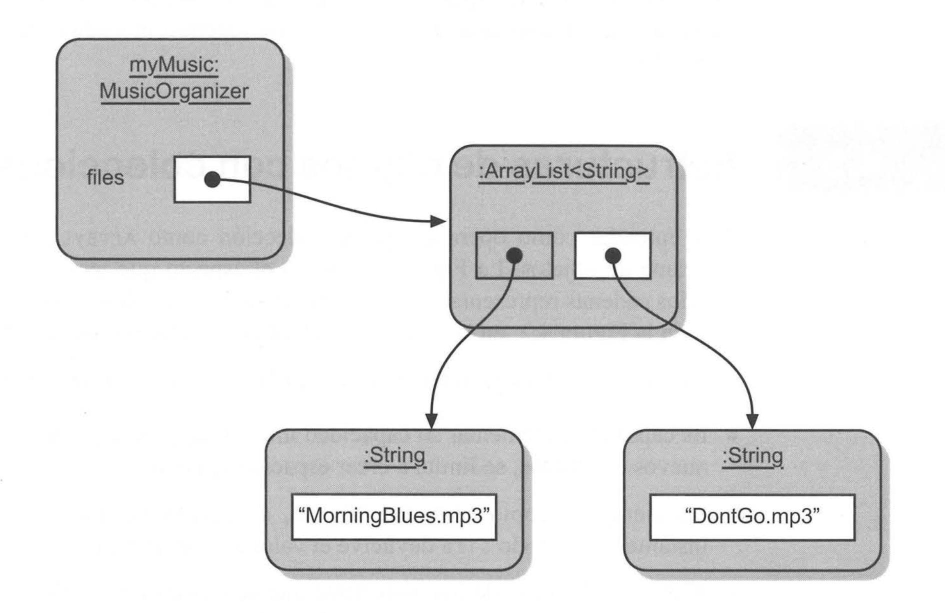


Figura 4.1
Un `MusicOrganizer` que contiene dos nombres de archivo.

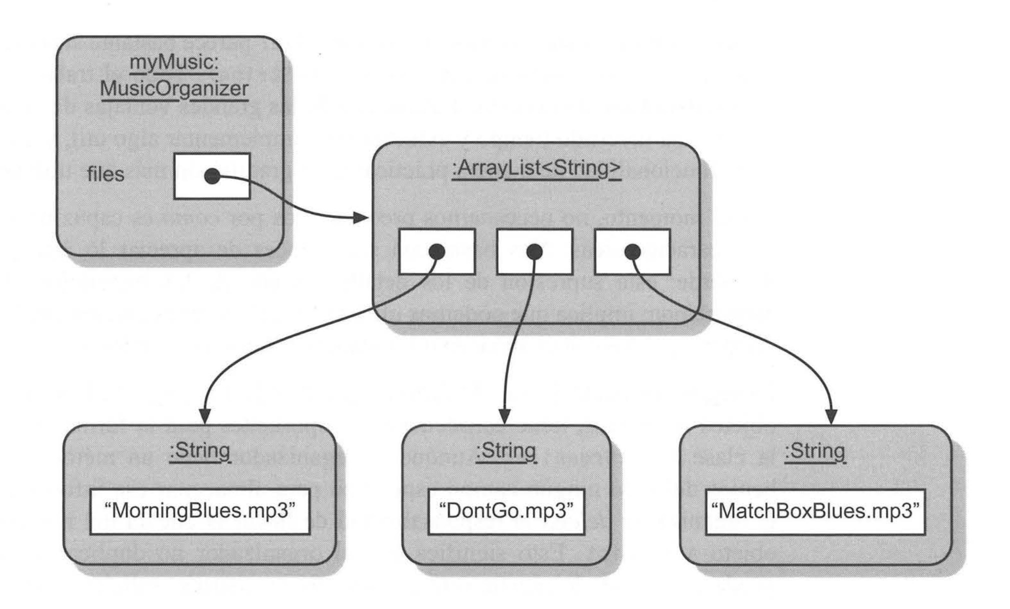


Figura 4.2
Un `MusicOrganizer` que contiene tres nombres de archivo.

4.6 Clases genéricas

La nueva notación que utiliza los corchetes angulares merece alguna explicación adicional. El tipo de nuestro campo `files` se ha declarado como

```
ArrayList<String>
```

La clase que estamos utilizando aquí se llama simplemente `ArrayList`, pero requiere que se especifique un segundo tipo como parámetro cuando se usa para declarar campos u otras variables. Las clases que requieren un parámetro de tipo como este se denominan *clases genéricas.* Las clases genéricas, a diferencia de otras clases que hemos visto hasta ahora, no definen un único tipo en Java, sino que pueden definir muchos tipos. Por ejemplo, la clase `ArrayList` puede emplearse para especificar una *colección ArrayList de objetos String*, una *colección ArrayList de objetos Person*, una *colección ArrayList de objetos Rectangle* o una colección `ArrayList` de objetos de cualquier otra clase que tengamos disponible. Cada `ArrayList` particular es un tipo distinto, que puede utilizarse en las declaraciones de campos, parámetros y valores de retorno. Por ejemplo, podríamos definir los dos campos siguientes:

```
private ArrayList<Person> members;
private ArrayList<TicketMachine> machines;
```

Estas definiciones indican que `members` hace referencia a un `ArrayList` que puede almacenar objetos `Person`, mientras que `machines` hace referencia a un `ArrayList` que se usa para almacenar objetos `TicketMachine`. Observe que `ArrayList<Person>` y `ArrayList<TicketMachine>` son tipos distintos. Los campos no pueden asignarse el uno al otro, aun cuando sus tipos deriven de la misma clase `ArrayList`.

Ejercicio 4.4 Escriba una declaración de un campo privado denominado `library` que pueda almacenar un `ArrayList`. Los elementos del `ArrayList` son de tipo `Book`.

Ejercicio 4.5 Escriba una declaración de una variable local denominada `cs101` que pueda almacenar un `ArrayList` de `Student`.

Ejercicio 4.6 Escriba una declaración de un campo privado denominado `tracks` para almacenar una colección de objetos `MusicTrack`.

Ejercicio 4.7 Escriba una serie de asignaciones a las variables `library`, `cs101` y `track` (que ha definido en los tres ejercicios anteriores) para crear los objetos `ArrayList` apropiados. Escriba esas asignaciones primero sin utilizar la notación diamante y después con la notación diamante, si es que está utilizando un compilador Java 7.

Las clases genéricas se utilizan para diversos propósitos; posteriormente en el libro veremos más acerca de ellas. Por el momento, las colecciones como `ArrayList`, y algunas otras colecciones que veremos enseguida, son las únicas clases genéricas que vamos a tener que manejar.

4.7 Numeración dentro de las colecciones

Al explorar el proyecto *music-organizer-v1* en los primeros ejercicios de este capítulo, dijimos que era necesario emplear valores de parámetro que comenzaran en 0 para visualizar y eliminar los nombres de archivo contenidos en la colección. La razón de este requisito es que los elementos almacenados en las colecciones `ArrayList` tienen una numeración o posicionamiento implícito que comienza en 0. La posición de un objeto dentro de una colección se conoce comúnmente como el nombre de *índice*. Al primer elemento añadido a una colección se le da el número de índice 0, al segundo se le da el número de índice 1, y así sucesivamente. La Figura 4.3 ilustra la misma situación anterior, pero con los números de índice especificados dentro del objeto `ArrayList`.

Es importante tener en cuenta que esto implica que el último elemento de una colección tiene como índice *size-1*. Por ejemplo, en una lista de 20 elementos, el último tendrá asociado el índice 19.

Los métodos `listFile` y `removeFile` ilustran la forma en que se utiliza el número de índice para acceder a un elemento de un `ArrayList`: uno lo hace a través del método `get` y el otro lo hace a través del método `remove`. Observe que ambos métodos se aseguran de que el valor de su parámetro se encuentre dentro del rango de valores de índice permitidos `[0...size()-1]` antes de pasar el índice a los métodos de `ArrayList`. Este es un buen hábito estilístico de validación a la hora de programar, ya que impide que falle una llamada a un método de una clase de librería, en aquellos casos en los que puedan pasarse valores de parámetro que pudieran ser no válidos.

Error común Si no tiene cuidado, podría terminar intentando acceder a un elemento de colección que se encuentre fuera del rango de índices válidos del objeto `ArrayList`. Si hace esto, obtendrá un mensaje de error y el programa terminará. Dicho error es del tipo *índice fuera de los límites*. En Java, podrá ver un mensaje acerca de una excepción `IndexOutOfBoundsException`.

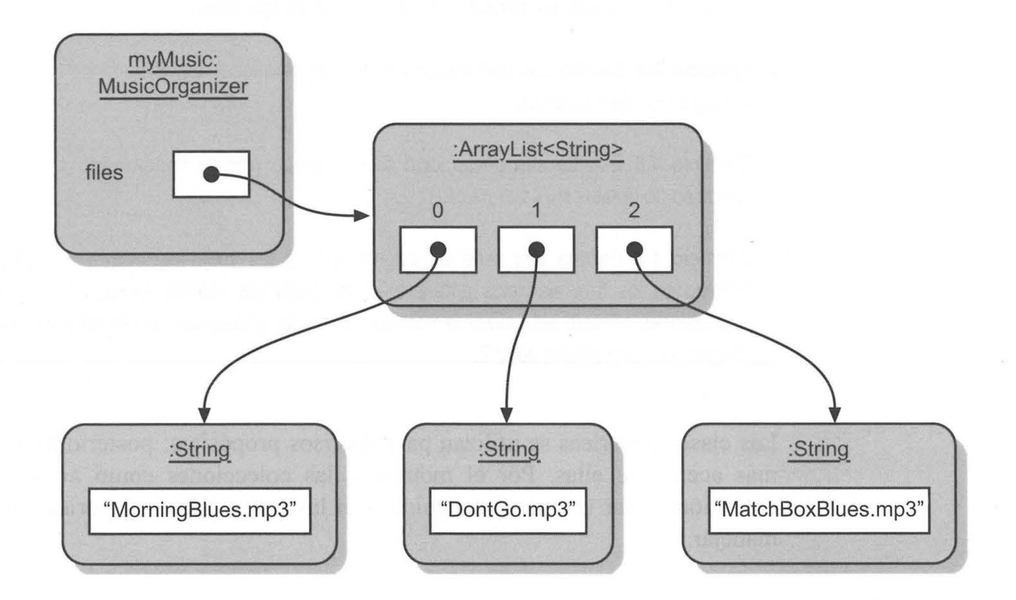


Figura 4.3
Números de índice de los elementos de una colección.

Ejercicio 4.8 Si una colección almacena 10 objetos, ¿qué valor devolverá una llamada a su método `size`?

Ejercicio 4.9 Escriba una llamada a método utilizando `get` para devolver el quinto objeto almacenado en una colección denominada `items`.

Ejercicio 4.10 ¿Cuál es el índice del último elemento almacenado en una colección de 15 objetos?

Ejercicio 4.11 Escriba una llamada a método para añadir a una colección denominada `files` el objeto almacenado en la variable `favoriteTrack`.

4.7.1 El efecto de las eliminaciones sobre la numeración

Además de añadir elementos a una colección, habitualmente también se desea poder eliminarlos, como hemos visto con el método `removeFile` en el Código 4.1. La clase `ArrayList` tiene un método `remove` que toma como parámetro el índice del objeto que hay que eliminar. Un detalle del proceso de eliminación del que debemos ser conscientes es que puede cambiar los valores de índice en los que están almacenados otros objetos de la colección. Si se elimina un objeto con un número de índice bajo, entonces la colección desplazará una posición todos los elementos posteriores con el fin de rellenar el hueco. Como consecuencia, sus números de índice disminuirán en una unidad.

La Figura 4.4 ilustra la forma en que se modifican algunos de los valores de índice de los elementos en un `ArrayList`, al eliminar un elemento situado en mitad de la lista. Partiendo de la situación mostrada en la Figura 4.3, se ha eliminado el objeto con índice 1; en consecuencia, el objeto situado originalmente en el índice número 2 ha cambiado a 1, mientras que el objeto en el número de índice 0 no sufre ninguna modificación.

Además, veremos posteriormente que es posible insertar elementos en un `ArrayList` en una posición distinta del final de la lista. Esto quiere decir que los elementos que ya se encuentren en la lista podrían ver sus números de índice incrementados cuando se añada un nuevo

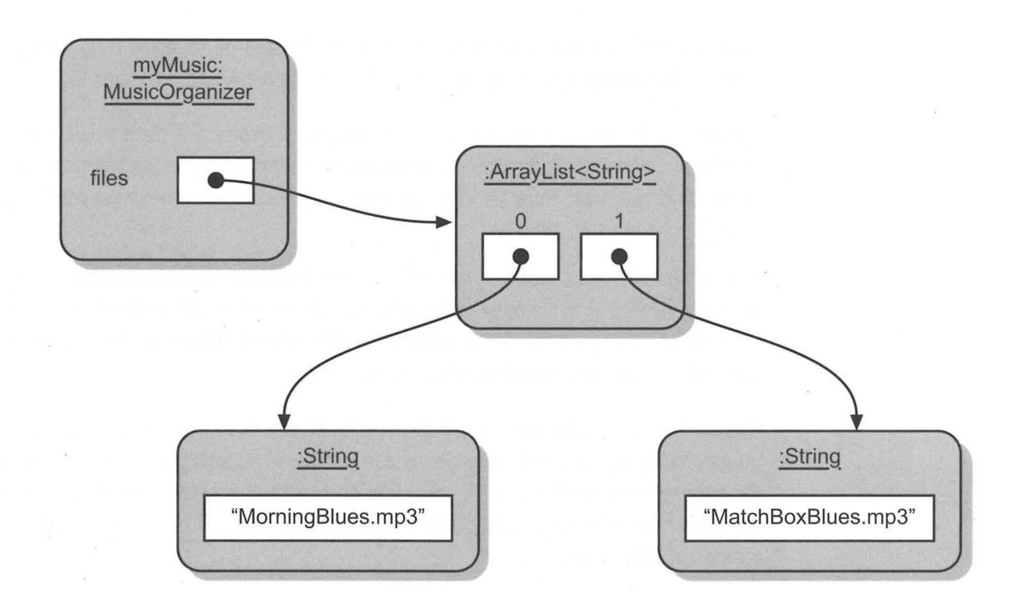


Figura 4.4 Cambios en los números de índice después de la eliminación de un elemento.

elemento. Los usuarios tienen que ser conscientes de estos posibles cambios de los índices a la hora de añadir o eliminar elementos.

4.7.2 Utilidad general de la numeración dentro de las colecciones

El uso de valores de índice enteros para acceder a los objetos de una colección es algo con lo que nos toparemos una y otra vez –no solo con colecciones `ArrayList`, sino también con otros tipos diferentes de colecciones. Por tanto, es importante entender lo que hemos visto hasta ahora en relación con esto: que los valores de índice comienzan en cero; que los objetos se numeran secuencialmente y que normalmente no suele haber huecos entre los valores de índice de los objetos consecutivos de la colección.

Utilizar valores enteros como índices hace también que sea muy fácil expresar en el código del programa conceptos tales como "el siguiente elemento" o "el elemento anterior", al hablar de un elemento de la colección. Si un elemento se encuentra en el índice `p`, entonces "el siguiente" se encontrará en el índice`(p+1)` y "el anterior" estará en el índice `(p-1)`. También se facilita la correspondencia entre selecciones hechas en lenguaje natural, como "los tres primeros" y la terminología relacionada con el programa. Por ejemplo, "los tres primeros" serían "los elementos con índices `0`, `1` y `2`", mientras que "los cuatro últimos" podrían ser "los elementos con índices comprendidos entre `(list.size()-4)` y `(list.size()-1)`". Podemos incluso imaginarnos recorriendo toda la colección mediante una variable entera utilizada como índice, cuyo valor se configure inicialmente con cero y que luego se incremente sucesivamente en una unidad, pasando dicho valor al método `get`, con el fin de acceder por orden a cada elemento de la lista (deteniéndonos cuando la variable sobrepase el valor final de índice de la lista).

Pero estamos adelantándonos a los acontecimientos. En cualquier caso, dentro de poco veremos cómo poner en práctica estos conceptos cuando examinemos los *bucles* y el mecanismo de *iteración*.

Ejercicio 4.12 Escriba una llamada a método para eliminar el tercer objeto almacenado en una colección denominada `dates`.

Ejercicio 4.13 Suponga que un objeto está almacenado en el índice 6 de una colección. ¿Cuál sería su índice inmediatamente después de eliminar los objetos con índice 0 e índice 9?

Ejercicio 4.14 Añada un método denominado `checkIndex` a la clase `MusicOrganizer`. Dicho método admite un único parámetro entero y comprueba si se trata de un índice válido, teniendo en cuenta el estado actual de la colección. Para ser válido, el parámetro tiene que estar comprendido en el rango que va de `0` a `size()-1`.

Si el parámetro no es valido, entonces debe imprimirse un mensaje de error que indique cuál es el rango válido de valores. Si el índice es válido, entonces el método no debe imprimir nada. Compruebe su método en el banco de objetos, introduciendo parámetros tanto válidos como inválidos. ¿Sigue funcionando su método al probarlo con un índice cuando la colección está vacía?

Ejercicio 4.15 Escriba una versión alternativa de `checkIndex` denominada `validIndex`. Debe tomar un parámetro entero y devolver un resultado booleano. No imprime nada, sino que devuelve *true* si el valor del parámetro es un índice válido, teniendo en cuenta el estado actual de la colección; en caso contrario, devuelve *false*. Pruebe su método en el banco de objetos con parámetros válidos e inválidos. Compruebe también el caso vacío.

Ejercicio 4.16 Escriba de nuevo los métodos `listFile` y `removeFile` de `MusicOrganizer` para que utilicen su método `validIndex` con el fin de comprobar su parámetro, en lugar de emplear la expresión booleana que aparece en el código. Los dos métodos nuevos solo deben invocar a `get` o `remove` sobre la colección `ArrayList` si `validIndex` devuelve *true*.

4.8 Reproducción de los archivos de música

Sería interesante que nuestro organizador fuera capaz no solo de mantener una lista de archivos de música, sino que también nos permitiera reproducirlos. De nuevo, podemos utilizar la abstracción como ayuda para resolver este problema. Si disponemos de una clase que haya sido escrita específicamente para reproducir archivos de audio, entonces nuestra clase para el organizador no necesitaría saber nada de cómo llevar a cabo esa tarea; se limitaría simplemente a pasar el nombre del archivo a la clase del reproductor y dejar que esta se encargara del resto.

Lamentablemente, la librería Java estándar no tiene una clase adecuada para reproducir archivos MP3, que es el formato de audio con el que queremos trabajar. Sin embargo, muchos programadores individuales están escribiendo continuamente sus propias clases de utilidad y poniéndolas a disposición de otras personas para que las utilicen. Estas se suelen denominar "librerías de terceras fuentes" y se importan y usan exactamente de la misma forma que las clases de la librería estándar de Java.

Para la siguiente versión de nuestro proyecto, hemos utilizado un conjunto de clases de `javazoom.net` con el fin de escribir nuestra propia clase reproductora de música. Puede encontrar dicha clase en la versión denominada *music-organizer-v2*. Los tres métodos de la clase `MusicPlayer` que utilizaremos son `playSample`, `startPlaying` y `stop`. Los dos primeros admiten el nombre del archivo de audio que hay que reproducir. El primero reproduce algunos segundos del principio del archivo y vuelve una vez que ha terminado de reproducirlos, mientras que el segundo comienza a reproducir en segundo plano y luego devuelve inmediatamente el control al organizador –de aquí la necesidad de emplear el método `stop`, para el caso de que queramos cancelar la reproducción. El Código 4.2 muestra los nuevos elementos de la clase `MusicOrganizer` que permiten acceder a parte de esta funcionalidad de reproducción.

Código 4.2
Funcionalidad de reproducción de la clase `MusicOrganizer`.

```
import java.util.ArrayList;

/**
 * Una clase para almacenar detalles de archivos de audio.
 * Esta versión puede reproducir los archivos.
 *
 * @author David J. Barnes y Michael Kölling
 * @version 2011.07.31
 */
public class MusicOrganizer
{
```

Código 4.2
(*continuación*)
Funcionalidad de reproducción de la clase `MusicOrganizer`.

```
    // Un ArrayList para almacenar los nombres de archivos de música.
    private ArrayList<String> files;
    // Un reproductor para los archivos de música.
    private MusicPlayer player;

     /**
      * Crear un MusicOrganizer
      */
     public MusicOrganizer()
     {
         files = new ArrayList<String>();
         player = new MusicPlayer();
     }

     /**
      * Comenzar a reproducir un archivo de la colección.
      * Utilice stopPlaying() para detener la reproducción.
      * @param index El índice del archivo que hay que reproducir.
      */
     public void startPlayingFile(int index)
     {
         String filename = files.get(index);
         player.startPlaying(filename);
     }

     /**
      * Detener el reproductor.
      */
     public void stopPlaying()
     {
         player.stop();
     }

     Se omiten otros detalles.
}
```

Ejercicio 4.17 Cree un objeto `MusicOrganizer` en la segunda versión de nuestro proyecto. Experimente añadiéndole algunos archivos y reproduciéndolos.

Si quiere utilizar los archivos proporcionados en la carpeta *audio* del proyecto, debe incluir el nombre de la carpeta en el parámetro `filename`, además de especificar el propio nombre de archivo y el sufijo. Por ejemplo, para usar el archivo *BlindBlake-EarlyMorningBlues.mp3* de la carpeta *audio*, tendrá que pasar la cadena `"audio/BlindBlake-EarlyMorningBlues.mp3"` al método `addFile`.

Puede utilizar sus propios archivos mp3 colocándolos en la carpeta *audio*. Acuérdese de usar el nombre de la carpeta como parte del nombre de archivo.

Experimente también especificando nombres de archivo que no existan. ¿Qué sucede al utilizarlos?

4.8.1 Resumen del organizador de música

Hemos progresado bastante con los fundamentos básicos de organización de nuestra colección de música. Podemos almacenar los nombres de cualquier número de archivos de música e incluso reproducirlos. Lo hemos hecho además con un esfuerzo de codificación relativamente pequeño, porque hemos podido aprovechar la funcionalidad proporcionada por las clases de librería: `ArrayList` de la librería estándar de Java y un reproductor de música de una librería de clases de una tercera fuente. También hemos podido hacerlo con un conocimiento relativamente pequeño del funcionamiento interno de estas clases de librería; nos ha bastado con conocer los nombres, los tipos de parámetros y los tipos de retorno de los métodos fundamentales.

Sin embargo, nos sigue faltando cierta funcionalidad clave si queremos disponer de un programa realmente útil –lo más obvio es la falta de, por ejemplo, una manera de enumerar toda la colección. Este será el tema de la siguiente sección, cuando presentemos la primera de una serie de estructuras de control de bucle en Java.

4.9 Procesamiento de una colección completa

Al final de la sección anterior, hemos dicho que sería útil de disponer de un método en el organizador de música que permitiera enumerar todos los nombres de archivo almacenados en la colección. Sabiendo que cada nombre de archivo de la colección tiene un número de índice distintivo, una forma expresar lo que deseamos sería decir que queremos mostrar el nombre de archivo almacenado en una serie de números de índice crecientes y consecutivos, comenzando por cero. Antes de continuar leyendo, realice los siguientes ejercicios para ver si podemos escribir fácilmente un método de ese tipo con el lenguaje Java que ya conocemos.

Ejercicio 4.18 ¿Qué aspecto podría tener la cabecera de un método `listAllFiles` utilizado para enumerar todos los archivos en la clase `MusicOrganizer`? ¿Qué tipo de retorno debería tener? ¿Necesita algún tipo de parámetro?

Ejercicio 4.19 Sabemos que el primer nombre de archivo está almacenado en el índice cero en el objeto `ArrayList` y que la lista almacena los nombres de archivo como cadenas de caracteres. En consecuencia, ¿podríamos escribir el cuerpo de `listAllFiles` con las siguientes instrucciones?

```
System.out.println(files.get(0));
System.out.println(files.get(1));
System.out.println(files.get(2));
```

etc. ¿Cuántas instrucciones `println` harían falta para completar el método?

Probablemente, se habrá dado cuenta de que no es posible completar el Ejercicio 4.19, porque depende de la cantidad de nombres de archivo que haya en la lista en el momento de imprimirlos. Si hubiera tres, harían falta tres instrucciones `println`; si hubiera cuatro, entonces se precisarían cuatro instrucciones; y así sucesivamente. Los métodos `listFile` y `removeFile` ilustran que el rango de números de índice válido en cualquier momento es `[0 a (size()-1)]`. Por tanto, cualquier método `listAllFiles` tendría que tener en cuenta también el tamaño dinámico para poder llevar a cabo su tarea.

Lo que necesitamos aquí es poder realizar una determinada acción varias veces, pero el número de veces depende de una serie de circunstancias que puede variar –en este caso, del tamaño de la colección. A la hora de programar, nos vamos a encontrar con este tipo de requerimiento en casi todos los programas que escribamos, y la mayoría de los lenguajes de programación tienen diversas formas de tratar con estos requerimientos, mediante el uso de *instrucciones de bucle*, que también se conocen con el nombre de *estructuras iterativas de control*.

El primer bucle que presentaremos para enumerar los archivos es un bucle especial que se utiliza con colecciones y que elimina completamente la necesidad de utilizar una variable de índice: se denomina *bucle for-each*.

4.9.1 El bucle for-each

Concepto:

Un **bucle** se puede utilizar para ejecutar un bloque de instrucciones repetidamente, sin tener que escribirlas múltiples veces.

Un *bucle for-each* es una de las formas de llevar a cabo repetidamente un conjunto de acciones sobre los elementos de una colección, pero sin tener que escribir dichas acciones más de una vez, lo que evita el problema con el que nos hemos encontrado en el Ejercicio 4.19. Podemos resumir la sintaxis Java y las acciones de un bucle for-each mediante el siguiente pseudocódigo:

```
for(TipoElemento elemento : colección) {
    cuerpo del bucle
}
```

El principal elemento nuevo de Java es la palabra `for`. El lenguaje Java tiene dos variantes del bucle `for`: una es el *bucle for-each*, que es el que estamos analizando aquí, y la otra se denomina simplemente *bucle for* y hablaremos de ella más adelante en el capítulo.

Un bucle for-each tiene dos partes: una cabecera del bucle (la primera línea de la instrucción de bucle) y un cuerpo de bucle situado a continuación de la cabecera. El cuerpo contiene aquellas instrucciones que queremos ejecutar una y otra vez.

El bucle for-each obtiene su nombre de la forma en que podemos interpretar su sintaxis: si leemos la palabra clave `for` como "*for each*" ("para cada") y los dos puntos de la cabecera del bucle como "*in*" ("en"), entonces la estructura del código mostrado anteriormente comienza a tener más sentido, como en este pseudocódigo (que no está escrito en Java):

```
for each elemento in colección do: {
    cuerpo del bucle
}
```

Cuando comparamos esta versión con el pseudocódigo original de la primera versión, podemos observar que *elemento* estaba escrito en la forma de una declaración de variable, como `TipoElemento elemento`. Esta sección declara de hecho una variable que luego se utiliza sucesivamente para cada elemento de la colección. Antes de seguir con nuestras explicaciones veamos un ejemplo real de código Java.

El Código 4.3 muestra una implementación de un método `listAllFiles` que enumera todos los nombres de archivo que se encuentran actualmente en el `ArrayList` del organizador. Para la enumeración, se utiliza un bucle for-each.

Código 4.3
Utilización de un bucle for-each para enumerar los nombres de archivo.

```
/**
 * Muestra una lista de todos los archivos de la colección.
 */
public void listAllFiles()
{
    for(String filename : files) {
        System.out.println(filename);
    }
}
```

En este bucle for-each, el cuerpo del bucle, compuesto por una única instrucción `System.out.println`, se ejecuta repetidamente, una vez por cada elemento contenido en el `ArrayList files`. Por ejemplo, si hubiera cuatro cadenas de caracteres en la lista, la instrucción `println` se ejecutaría cuatro veces.

Cada vez, antes de ejecutar la instrucción, se almacena en la variable `filename` uno de los elementos de la lista: primero el que se encuentra en el índice `0`, después el del índice `1`, etc. De este modo, terminan imprimiéndose todos los elementos de la lista.

Analicemos el bucle con un poco más de detalle. La palabra clave `for` inicia el bucle. Va seguida de una pareja de paréntesis, dentro de los cuales se definen los detalles del bucle. El primero de esos detalles es la declaración `String filename`, que declara una nueva variable local `filename` que se utilizará para almacenar sucesivamente los distintos elementos de la lista. A esta variable la denominados *variable de bucle*. Podemos elegir el nombre que queramos para esta variable, al igual que sucede con cualquier otra variable; no tenemos por qué denominarla *filename*. El tipo de la variable de bucle debe coincidir con el tipo de elemento declarado para la colección que vayamos a utilizar; en este caso, `String`.

A continuación, aparece un carácter de dos puntos y luego la variable que contiene la colección que queremos procesar. Para esta colección, cada elemento será asignado por turno a la variable de bucle; y para cada una de esas asignaciones, se ejecuta una vez el cuerpo del bucle. En el cuerpo del bucle, utilizamos la variable de bucle para hacer referencia a cada elemento.

Con el fin de comprobar si comprende cómo funciona este bucle, pruebe a hacer los siguientes ejercicios.

Ejercicio 4.20 Implemente el método `listAllFiles` en su versión del proyecto music-organizer. (En la versión *music-organizer-v3* de este proyecto se proporciona una solución con este método implementado, pero le recomendamos que escriba usted mismo este método para comprender mejor los conceptos.)

Ejercicio 4.21 Cree un objeto `MusicOrganizer` y almacene en él unos cuantos nombres de archivo. Utilice el método `listAllFiles` para imprimirlos; compruebe que el método funciona correctamente.

Ejercicio 4.22 Cree una colección `ArrayList<String>` en el Code Pad escribiendo las dos líneas siguientes:

```
import java.util.ArrayList;
new ArrayList<String>()
```

Si escribe la última línea sin añadir un punto y coma al final, aparecerá el pequeño icono rojo de objeto. Arrastre este icono hasta el banco de objetos. Examine sus métodos y pruebe a invocar algunos de ellos (como por ejemplo `add`, `remove`, `size`, `isEmpty`). Pruebe también a invocar los mismos métodos desde el Code Pad. Puede acceder a los objetos del banco de objetos desde el Code Pad utilizando sus nombres. Por ejemplo, si tiene un `ArrayList` denominado `a11` en el banco de objetos, puede escribir en el Code Pad:

```
a11.size()
```

Ejercicio 4.23 Si lo desea, puede emplear el depurador para tratar de comprender mejor cómo se repiten las instrucciones contenidas en el cuerpo del bucle de `listAllFiles`. Defina un punto de interrupción justo antes del bucle y ejecute paso a paso el método hasta que el bucle haya procesado todos los elementos y termine.

Ejercicio 4.24 *Ejercicio avanzado* El bucle for-each no utiliza una variable entera explícita para acceder a los elementos sucesivos de la lista. Por tanto, si queremos incluir el índice de cada nombre de archivo en el listado, entonces tendremos que declarar nuestra propia variable entera local (por ejemplo, `position`) para poder escribir en el cuerpo del bucle algo así como:

```
System.out.println(position + ": " + filename);
```

Compruebe si puede completar una versión de `listAllFiles` para hacer esto. *Sugerencia:* necesitará una declaración de variable local para `position` en el método, así como una instrucción para incrementar su valor en una unidad dentro del bucle for-each.

Una de las cosas que este ejercicio ilustra es que el bucle for-each no está pensado realmente para utilizarse con una variable de índice independiente.

Con esto hemos visto cómo se puede utilizar un bucle for-each para realizar algunas operaciones (el cuerpo del bucle) con cada elemento de una colección. Esto representa un gran paso adelante, pero no resuelve todos nuestros problemas. En ocasiones, necesitamos algo más de control, y Java proporciona otra estructura de bucle distinta que nos permitirá hacer más cosas: el bucle *while*.

4.9.2 Procesamiento selectivo de una colección

El método `listAllFiles` ilustra la utilidad fundamental de un bucle for-each: proporciona acceso a cada elemento de una colección, por orden, a través de la variable declarada en la cabecera del bucle. No nos proporciona la posición de índice de cada elemento, pero no siempre necesitamos dicha posición de índice, así que eso no constituye necesariamente un problema.

Sin embargo, tener acceso a todos los elementos de una colección no implica que tengamos que llevar a cabo las mismas acciones cada vez; podemos ser bastante más selectivos. Por ejemplo, podríamos querer enumerar únicamente las canciones de un artista concreto o encontrar todos los archivos de música que tengan una determinada frase en el título. No hay nada que nos impida hacer esto, porque el cuerpo de un bucle for-each es un bloque común y corriente, y podemos utilizar dentro de él cualquier instrucción Java que queramos. De ese modo, es sencillo emplear una instrucción if en el cuerpo del bucle para seleccionar los archivos que deseemos.

El Código 4.4 muestra un método que enumera únicamente aquellos nombres de archivo de la colección que contienen una cadena de caracteres concreta.

Código 4.4
Impresión de elementos seleccionados de la colección.

```
/**
 * Enumera los nombres de archivo que se corresponden con
 * la cadena de búsqueda proporcionada.
 * @param searchString La cadena que hay que buscar.
 */
public void listMatching(String searchString)
{
    for(String filename : files) {
        if(filename.contains(searchString)) {
            // Una coincidencia.
            System.out.println(filename);
        }
    }
}
```

Utilizando una instrucción `if` y el resultado `boolean` del método `contains` de la clase `String`, podemos "filtrar" los nombres de archivo que hay que imprimir y los que no. Si el nombre de archivo no se corresponde, nos limitamos a ignorarlo –no hace falta ninguna parte `else`. El criterio de filtrado (la comprobación en la instrucción if) puede ser cualquiera que deseemos.

Ejercicio 4.25 Añada el método `listMatching` del Código 4.4 a su versión del proyecto. (Utilice *music-organizer-v3* si no dispone todavía de su propia versión.) Compruebe que el método enumera únicamente los archivos que se corresponden con el criterio de búsqueda. Pruebe también el método con una cadena de búsqueda que no se corresponda con ninguno de los nombres de archivo. ¿Llega a imprimirse algo en este caso?

Ejercicio 4.26 *Ejercicio avanzado* En `listMatching`, ¿sería capaz de encontrar una manera de imprimir un mensaje, después de terminado el bucle for-each, si no se ha encontrado ningún nombre de archivo que se corresponda con la cadena de búsqueda? *Sugerencia:* utilice una variable local `boolean`.

Ejercicio 4.27 Escriba un método en su versión del proyecto que reproduzca muestras de todas las canciones de un artista concreto, una después de otra. El método `listMatching` ilustra la estructura básica que le hará falta para este método. Asegúrese de elegir un artista que tenga más de un archivo almacenado. Utilice el método `playAndWait` del `MusicPlayer`, en lugar del método `startPlaying`; en caso contrario, terminará reproduciendo al mismo tiempo todas las canciones que se correspondan con el criterio de búsqueda. El método `playAndWait` reproduce el principio de una canción (unos 15 segundos) y luego vuelve.

Ejercicio 4.28 Escriba la cabecera de un bucle for-each para procesar una colección `ArrayList<Track>` denominada `tracks`. No se preocupe del cuerpo del bucle.

4.9.3 Una limitación en el uso de cadenas de caracteres

Por supuesto, llegados a este punto podemos ver que el disponer simplemente de cadenas con nombres de archivo que contienen todos los detalles de cada canción no es una solución

realmente satisfactoria. Por ejemplo, suponga que queremos encontrar todas las canciones con la palabra *love* en el título. Utilizando la técnica de búsqueda poco sofisticada que hemos descrito anteriormente, también localizaríamos las canciones de artistas cuyos nombres resulten contener dicha secuencia de caracteres (por ejemplo, Glover). Aunque esto no parece un problema realmente grave, hace que el programa parezca "chapucero" y está claro que deberíamos poder conseguir una solución mejor con un pequeño esfuerzo adicional. Lo que realmente necesitamos es una clase independiente, por ejemplo, `Track`, que almacene los detalles del artista y del título de forma separada del nombre de archivo. De esta forma, podemos realizar búsquedas en los títulos de manera independiente de las búsquedas en los nombres de autor. Entonces, sustituiríamos la colección `ArrayList<String>` del organizador por otra colección `ArrayList<Track>`.

A medida que desarrollemos el proyecto del organizador de música en las siguientes secciones, llegaremos a definir una estructura más adecuada, introduciendo una clase `Track`.

4.9.4 Resumen del bucle for-each

El bucle for-each se emplea siempre para iterar a través de una colección. Nos proporciona una forma de acceder a cada elemento de la colección sucesivamente, uno por uno, y procesar esos elementos de la forma que deseemos. Podemos decidir llevar a cabo las mismas acciones con cada elemento (como hicimos al imprimir la lista completa) o podemos ser selectivos y filtrar la lista (como hicimos al imprimir solo un subconjunto de la colección). El cuerpo del bucle puede ser todo lo complicado que queramos.

Pero esta simplicidad esencial lleva aparejadas, necesariamente, algunas limitaciones. Por ejemplo, una restricción es que no podemos modificar lo que está almacenado en la colección mientras iteramos a través de ella: no podemos ni añadir nuevos elementos ni eliminar elementos de la colección. Aunque esto no significa, sin embargo, que no podamos cambiar el estado de los objetos que ya están dentro de la colección.

También hemos visto que el bucle for-each no nos proporciona un valor de índice para los elementos de la colección. Si deseamos uno, tendremos que declarar y mantener nuestra propia variable local. La razón de esto tiene que ver, de nuevo, con la abstracción. Al tratar con colecciones e iterar a través de ellas, resulta útil tener presentes dos consideraciones:

- Un bucle for-each proporciona una estructura general de control para iterar a través de diferentes tipos de colecciones.
- Existen algunos de tipos de colecciones que no asocian de manera natural índices enteros con los elementos que almacenan. Nos encontraremos con algunas de estas colecciones en el Capítulo 5.

Por tanto, el bucle for-each abstrae la tarea de procesar una colección completa elemento a elemento y es capaz de manejar diferentes tipos de colección. No necesitamos saber los detalles de cómo manipula las colecciones.

Una de las cuestiones que no nos hemos planteado es si puede utilizarse un bucle for-each en aquellos casos en los que deseemos terminar el procesamiento en mitad de una colección. Por ejemplo, suponga que en lugar de reproducir todas las canciones de un cierto artista, simplemente queremos encontrar la primera y reproducirla, sin continuar con las otras.

Aunque en principio es posible hacer esto utilizando un bucle for-each, le recomendamos que no emplee bucles for-each para aquellas tareas en las que pueda no ser necesario procesar la colección completa. En otras palabras, le recomendamos emplear un bucle for-each solo si está seguro de querer procesar la *colección completa.* Dicho de otra forma, una vez que el bucle comience, sabremos con seguridad cuántas veces se va a ejecutar el cuerpo del bucle –ese número de veces será igual al tamaño de la colección. Este estilo se denomina en ocasiones *iteración definida.* Para aquellas tareas en las que queramos detener anticipadamente el procesamiento de la colección, hay otros bucles más apropiados que se pueden utilizar –por ejemplo, el *bucle while*, que presentaremos a continuación. En estos casos, el número de veces que se ejecutará el cuerpo del bucle es menos preciso; normalmente, dependerá de lo que suceda durante la iteración. Este estilo se denomina en ocasiones *iteración indefinida* y es lo que vamos a analizar a continuación.

4.10 Iteración indefinida

La utilización de un bucle for-each nos ha proporcionado nuestra primera experiencia con el concepto de llevar a cabo una serie de acciones de manera repetida. Las instrucciones contenidas en el cuerpo del bucle se repiten para cada elemento de la colección asociada, y la iteración se detiene cuando alcanzamos el final de la colección. Un bucle for-each proporciona una *iteración definida*; dado el estado de una colección concreta, el cuerpo del bucle se ejecutará un número de veces que se corresponde exactamente con el tamaño de dicha colección. Pero hay muchas situaciones en las que queremos repetir una serie de acciones, pero en las que no podemos predecir de antemano exactamente cuántas veces va a ser. Un bucle for-each no nos sirve de ayuda en estos casos.

Imagine, por ejemplo, que ha perdido las llaves y que necesita encontrarlas antes de salir de casa. El proceso de búsqueda se podría modelar con una iteración indefinida, porque hay muchos lugares distintos en los que buscar y no podemos predecir de antemano en cuántos lugares buscaremos antes de encontrar las llaves; después de todo si pudiéramos predecir eso ¡iríamos directamente al lugar donde se encuentran las llaves! Así que lo que haremos será componer mentalmente una lista de lugares posibles en los que puedan estar las llaves y luego visitar cada uno de esos sitios sucesivamente, hasta encontrarlas. Una vez encontradas, lo que haremos será detenernos en lugar de completar la lista (lo que no tendría ningún sentido).

Lo que tenemos aquí es un ejemplo de *iteración indefinida*: la acción (la búsqueda) se repetirá un número de veces no predecible hasta que se complete la tarea. En programación, es bastante común encontrarse con escenarios similares al de la búsqueda de las llaves. Aunque no siempre vamos a estar buscando algo, frecuentemente nos vamos a encontrar con situaciones en las que querremos seguir haciendo una determinada cosa hasta que la repetición deje de ser necesaria. De hecho, estas situaciones son tan comunes que la mayoría de los lenguajes de programación proporcionan al menos una (y normalmente más de una) estructura de bucle para expresarlas. Como lo que intentamos hacer con esas estructuras de bucle suele ser más complejo que el limitarse a iterar a través de una colección completa desde el principio hasta al final, esas estructuras son algo más difíciles de comprender, pero el esfuerzo de compresión necesario se verá más que recompensado por la gran variedad de cosas que vamos a poder hacer con ellas. Nuestra atención aquí se centrará en el *bucle while* de Java, que es similar a los bucles que podemos encontrar en otros lenguajes de programación.

4.10.1 El bucle while

Un *bucle while* consta de una cabecera y de un cuerpo; el cuerpo está pensado para ser ejecutado de manera repetida. He aquí la estructura de un bucle `while` donde *condición booleana* y *cuerpo del bucle* son pseudocódigo, pero todo lo restante es la sintaxis de Java:

```
while (condición booleana) {
    cuerpo del bucle
}
```

El bucle se inicia con la palabra clave `while`, que va seguida de una condición booleana. La condición es la que controla, en último término, cuántas veces se iterará un bucle concreto. La condición se evalúa cuando el control del programa alcanza por primera vez el bucle, y vuelve a evaluarse después de ejecutar cada vez el cuerpo del bucle. Esto es lo que da al bucle while su carácter indefinido –ese proceso de re-evaluación. Si la condición se evalúa como *true*, entonces se ejecuta el cuerpo del bucle; y una vez que la condición se evalúe como *false*, se da por terminada la iteración. Entonces el programa se salta el cuerpo del bucle y la ejecución continúa con lo que quiera que haya a continuación del bucle. Observe que la condición podría llegar a evaluarse como *false* la primera vez que se comprueba. Si sucediera esto, el cuerpo del bucle no llegaría a ejecutarse nunca. Esta es una característica muy importante del bucle while: el cuerpo del bucle puede ejecutarse cero veces, en lugar de ejecutarse siempre al menos una vez.

Antes de ver un ejemplo Java real, echemos un vistazo a una versión en pesudocódigo del proceso de búsqueda de llaves que hemos descrito anteriormente, para tratar de comprender cómo funciona un bucle while. He aquí una manera de expresar el proceso de búsqueda:

```
while(nos faltan las llaves) {
      buscar en el siguiente sitio
}
```

Cuando llegamos al bucle por primera vez, se evalúa la condición y comprobamos que no tenemos las llaves. Esto quiere decir que entramos en el cuerpo del bucle y buscamos en el siguiente lugar enumerado en nuestra lista mental. Habiendo hecho eso, volvemos a la condición y la evaluamos de nuevo. Si hemos encontrado las llaves, se da por terminado el bucle y nos lo podemos saltar y salir de casa. Si nos siguen faltando las llaves, volvemos a entrar en el cuerpo del bucle y buscamos en el siguiente sitio. Este proceso repetitivo continuará hasta que ya no nos falten las llaves.[1]

Observe que también podríamos haber expresado la condición del bucle de la forma contraria, como se indica a continuación:

```
while(not (hemos encontrado las llaves)) {
      buscar en el siguiente sitio
}
```

La distinción es sutil –en un caso expresamos la condición como un estado que hay que cambiar, mientras que en el otro la expresamos como un objetivo que aun no ha sido alcanzado. Tómese su tiempo para leer las dos versiones, con el fin de asegurarse de que comprende cómo

---

[1] En esta etapa, vamos a ignorar la posibilidad de que no lleguemos nunca a encontrar las llaves, pero tener en cuenta este tipo de posibilidad tendrá una gran importancia cuando examinemos ejemplos Java reales.

funciona cada una. Ambas son igualmente válidas, y reflejan las decisiones sobre como expresar las condiciones que tendremos que tomar a la hora de escribir bucles reales. En ambos casos, lo que escribimos dentro del bucle al encontrar finalmente las llaves implicará que las condición del bucle "cambie" de *true* a *false* la siguiente vez que sea evaluada.

Ejercicio 4.29 Suponga que expresamos en pesudocódigo la primera versión del proceso de búsqueda de llaves de la manera siguiente:

```
boolean missing = true;
while(missing) {
    if(las llaves están en el siguiente lugar) {
        missing = false;
    }
}
```

Trate de expresar la segunda versión completando el siguiente esquema:

```
boolean found = false;
while(...) {
    if(las llaves están en el siguiente lugar) {
        ...
    }
}
```

4.10.2 Iteración mediante una variable de índice

Para nuestro primer bucle while en código Java correcto, escribiremos una versión del método `listAllFiles` mostrado en el Código 4.3. Esto no ilustra realmente el carácter indefinido de los bucles while, pero proporciona una comparación útil con el familiar ejemplo equivalente basado en for-each. La versión con bucle while se muestra en el Código 4.5. Una característica clave es la forma en que se utiliza una variable entera (`index`) tanto para acceder a los elementos de la lista como para controlar la longitud de la iteración.

Código 4.5 Utilización de un bucle while para enumerar todas las canciones.

```
/**
 * Mostrar una lista de todos los archivos de
 * la colección.
 */
public void listAllFiles()
{
    int index = 0;
    while(index < files.size()) {
        String filename = files.get(index);
        System.out.println(filename);
        index++;
    }
}
```

Es inmediatamente obvio que la versión con el bucle while requiere más esfuerzo de programación por nuestra parte. Observe lo siguiente:

- Tenemos que declarar una variable para emplearla como índice de la lista y tenemos que inicializarla con el valor 0 para acceder al primer elemento de la lista. La variable tiene que declararse fuera del bucle.
- Tenemos que resolver cómo expresar la condición del bucle para garantizar que el bucle se detenga en el momento correcto.
- Los elementos de la lista no se extraen automáticamente de la colección ni se asignan automáticamente a una variable. En lugar de ello, tenemos que hacerlo nosotros mismos, utilizando el método `get` de `ArrayList`. La variable `filename` será local al cuerpo del bucle.
- Tenemos que acordarnos de incrementar la variable contador (`index`) nosotros mismos, para garantizar que la condición del bucle llegue a ser en algún momento *false* cuando hayamos alcanzado el final de la lista.

La instrucción final del cuerpo del bucle while ilustra un operador especial utilizado para incrementar en 1 una variable numérica:

```
index++;
```

Esta instrucción es equivalente a

```
index = index + 1;
```

Hasta ahora, el bucle for-each resulta claramente más elegante para nuestros propósitos. Era menos complicado de escribir y resulta más seguro. La razón de que sea más seguro es que siempre se garantiza que el bucle termine. En nuestra versión con el bucle while, es posible cometer un error que haga que tengamos un *bucle infinito*. Si nos olvidamos de incrementar la variable `index` (como se hace en la última línea del cuerpo del bucle), la condición del bucle nunca llegaría a ser *false* y el bucle seguiría iterando indefinidamente. Este es un error de programación típico que hasta los programadores más expertos cometen alguna que otra vez. El programa se ejecutará entonces indefinidamente. Si el bucle no contiene, en esa situación, ninguna instrucción de salida, el programa parecerá haberse "colgado", parece que no hace nada y no responde a los clics del ratón o las pulsaciones de tecla. En realidad, el programa está haciendo muchas cosas: está ejecutando el bucle una y otra vez, pero lo que sucede es que no podemos ver que esto tenga algún efecto, así que el programa parece haberse muerto. En BlueJ, esto puede detectarse a menudo por el hecho de que el indicador de "ejecución" a rayas rojas y blancas permanece activado, mientras que el programa parece no estar haciendo nada.

Así que, ¿cuáles son los beneficios de un bucle while con respecto a un bucle for-each? Hay dos clases de ventajas: en primer lugar, el bucle while no necesita estar relacionado con una colección (podemos construir un bucle con cualquier condición que podamos escribir en forma de expresión booleana); en segundo lugar, aunque estemos utilizando el bucle para procesar una colección, es posible que no necesitemos procesar todos los elementos –en lugar de ello, podríamos detenernos anticipadamente, si así lo deseamos, incluyendo otra componente dentro de la condición del bucle que exprese por qué querríamos terminar el bucle. Por supuesto, estrictamente hablando, lo que la condición del bucle expresa en realidad es por qué querríamos continuar, y es la negación de esa condición la que hace que el bucle se detenga.

Una ventaja de tener una variable de índice explícita es que podemos utilizar su valor tanto dentro como fuera del bucle, lo que no podíamos hacer en los ejemplos de for-each. Así, podemos incluir el índice en el listado si queremos. Esto nos facilitará la tarea de seleccionar una canción según su posición en la lista. Por ejemplo:

```
int index = 0;
while(index < files.size()) {
    String filename = files.get(index);
    // Utilizar el índice de la canción como prefijo del nombre de archivo.
    System.out.println(index + ": " + filename);
    index++;
}
```

El tener una variable de índice local puede ser particularmente importante al realizar búsquedas en una lista, porque puede proporcionar información sobre dónde estaba ubicado el elemento y podemos hacer que esa información siga estando disponible una vez que el bucle haya finalizado. Veremos esto en la siguiente sección.

4.10.3 Búsquedas en una colección

Las búsquedas son una de las formas más importantes de iteración con las que nos vamos a encontrar. Es fundamental, por tanto, comprender adecuadamente sus elementos esenciales. Los tipos de estructuras de bucle empleados en las búsquedas aparecen una y otra vez en situaciones reales de programación.

La característica clave de una búsqueda es que implica una *iteración indefinida*; esto tiene que ser necesariamente así, porque si supiéramos exactamente dónde buscar, ¡no nos haría falta realizar una búsqueda en absoluto! En lugar de ello, lo que tenemos que hacer es iniciar una búsqueda y luego nos hará falta un número desconocido de iteraciones antes de completarla. Esto implica que un bucle for-each es inapropiado para las búsquedas, porque siempre llevará a cabo su conjunto completo de iteraciones.[2]

En situaciones reales de búsqueda, tenemos que tener en cuenta que la búsqueda puede fallar: es posible que nos quedemos sin lugares en los que buscar. Eso quiere decir que normalmente tendremos que tomar en consideración dos posibilidades de finalización a la hora de escribir un bucle de búsqueda:

- La búsqueda tiene éxito después de un número indefinido de iteraciones.
- La búsqueda falla después de agotar todas las posibilidades.

Debemos tener en cuenta las dos posibilidades a la hora de escribir la condición del bucle. Como la condición del bucle debe evaluarse como *true* si queremos iterar otra vez más, cada uno de los criterios de finalización debe poder hacer, por sí mismo, que la condición se evalúe como *false* para detener el bucle.

El hecho de que terminemos analizando la lista completa en aquellos casos en los que la búsqueda falla no hace que las búsquedas fallidas constituyan un ejemplo de iteración definida.

---

[2] Aunque existen formas de alterar esta característica de un bucle for-each, y aunque esas formas se emplean de manera bastante común, a nuestro juicio se trata de un mal estilo de programación y no vamos a usarlas en nuestros ejemplos.

La característica clave de la iteración definida es que podemos determinar el número de iteraciones en el *momento de iniciarse el bucle*. Ese no será nunca el caso cuando estemos haciendo una búsqueda.

Si estamos empleando una variable de índice para ir avanzando a través de elementos sucesivos de una colección, entonces las búsquedas fallidas son fáciles de identificar: la variable de índice se habrá incrementado más allá del índice correspondiente al elemento final de la lista. Esa es exactamente la situación cubierta en el método `listAllFiles` del Código 4.5, donde la condición es:

```
while(index < files.size())
```

La condición expresa que queremos continuar siempre y cuando el índice se encuentre dentro del rango de índice válidos de la colección; en cuanto se incremente y salga del rango, queremos que el bucle se detenga. Merece la pena resaltar que esta condición funciona incluso aunque la lista esté vacía. En este caso, `index` habrá sido inicializada con el valor cero y la llamada a la método `size` devolverá también cero. Puesto que cero no es menor cero, el cuerpo del bucle no llegará a ejecutarse, que es lo que queremos.

También necesitamos añadir una segunda parte a la condición, que indique si hemos encontrado ya el elemento de búsqueda y que detenga la búsqueda en caso de que lo hayamos hecho. Hemos visto en la Sección 4.10.1 y en el Ejercicio 4.29 que a menudo podemos expresar esta condición de manera positiva o negativa, mediante variables booleanas apropiadamente inicializadas:

- Una variable denominada `searching` (o por ejemplo `missing`) configurada inicialmente con el valor *true* haría que la búsqueda continuara hasta que la variable se configurara como *false* dentro del bucle, después de haber encontrado el elemento.
- Una variable denominada `found`, configurada inicialmente como *false* y utilizada en la condición como `!found` haría que la búsqueda continuara hasta que la variable se configurara como *true* después de encontrar el elemento.

He aquí los dos fragmentos de código correspondientes que expresan la condición completa en ambos casos:

```
int index = 0;
boolean searching = true;
while(index < files.size() && searching)
```

y

```
int index = 0;
boolean found = false;
while(index < files.size() && !found)
```

Tómese su tiempo para asegurarse de que comprende estos dos fragmentos, que implementan ambos el mismo tipo exacto de control del bucle, pero expresado de forma ligeramente distinta. Recuerde que *toda la* condición debe evaluarse como *true* si queremos continuar buscando, y que debe evaluarse como *false* si queremos dejar de buscar, por la razón que sea. En el Capítulo 3 hemos hablado ya del operador "and" `&&`, que solo se evalúa como *true* si *los dos* operandos son *true*.

En el Código 4.6 (*music-organizer-v4*) se muestra la versión completa de un método para buscar el primer nombre de archivo que se corresponda con una cadena de búsqueda

determinada. El método devuelve el índice del elemento como resultado. Observe que necesitamos una forma de indicar a quien haya invocado el método si la búsqueda ha fallado. En este caso, lo que hacemos es devolver un valor que es imposible que represente una ubicación válida dentro de la colección –un valor negativo. Esta es una técnica comúnmente utilizada en las búsquedas: devolver un valor *fuera de límites* para indicar que la búsqueda ha fallado.

Código 4.6
Localización del primer elemento de una lista que se corresponde con un criterio de búsqueda.

```
/**
 * Localizar el índice del primer archivo que se corresponde con
 * la cadena de búsqueda indicada.
 * @param searchString La cadena que hay que buscar.
 * @return El índice de la primera aparición, es decir -1 si
 *         no se encuentra ninguna correspondencia.
 */
public int findFirst(String searchString)
{
    int index = 0;
    /* Indicar que vamos a seguir buscando hasta encontrar
     * una correspondencia.
     */
    boolean searching = true;

    while(searching && index < files.size()) {
        String filename = files.get(index);
        if(filename.contains(searchString)) {
            // Una correspondencia. Podemos dejar de buscar.
            searching = false;
        }
        else {
            // Pasar al siguiente elemento.
            index++;
        }
    }
    if(searching) {
        // No la hemos encontrado.
        return -1;
    }
    else {
        // Devolver la ubicación donde la hayamos encontrado.
        return index;
    }
}
```

Es tentador tratar de tener una única condición en el bucle, aunque existan dos razones diferentes para detener la búsqueda. Una forma de hacer esto sería modificar artificialmente el valor de `index` para que fuera demasiado grande en caso de que encontráramos lo que estamos buscando. Esta es una práctica que desaconsejamos, porque hace que el criterio de terminación del bucle sea confuso y siempre es preferible la claridad a la hora de programar.

4.10.4 Algunos ejemplos no relacionados con colecciones

Los bucles no se utilizan solo con colecciones. Existen muchas situaciones en las que queremos repetir un bloque de instrucciones en el que no interviene para nada ninguna colección. He aquí un ejemplo que imprime todos los número pares comprendidos entre 0 y 30:

```
int index = 0;
while(index <= 30) {
    System.out.println(index);
    index = index + 2;
}
```

De hecho, en este caso estamos utilizando un bucle while para una iteración definida, porque está claro desde el principio cuántos números vamos a imprimir. Sin embargo, no podemos emplear un bucle for-each, porque esos bucles solo pueden usarse para iterar a través de colecciones. Posteriormente, nos encontraremos con un tercer bucle relacionado con los anteriores –el *bucle for*– que resulta más apropiado para este ejemplo concreto.

Para comprobar que comprende cómo se usan los bucles while en aquellos casos en los que no interviene una colección, trate de completar los siguientes ejercicios.

Ejercicio 4.30 Escriba un bucle while (por ejemplo, en un método denominado `multiplesOfFive`) que imprima todos los múltiplos de 5 comprendidos entre 10 y 95.

Ejercicio 4.31 Escriba un bucle while para sumar los valores comprendidos entre 1 y 10 e imprimir la suma después de que el bucle haya finalizado.

Ejercicio 4.32 Escriba un método denominado `sum` con un bucle while que sume todos los números comprendidos entre dos números `a` y `b`. Los valores de `a` y `b` pueden pasarse al método `sum` como parámetros.

Ejercicio 4.33 *Ejercicio avanzado* Escriba un método `isPrime(int n)` que devuelva *true* si el parámetro `n` es un número primo y *false* si no lo es. Para implementar el método, puede escribir un bucle while que divida `n` entre todos los números comprendidos entre `2` y `(n-1)` y compruebe si la división da un número entero. Puede escribir esta comprobación utilizando el operador módulo (`%`) para ver si la división entera deja un resto igual a 0 (consulte las explicaciones acerca del operador módulo en la Sección 3.8.3).

Ejercicio 4.34 En el método `findFirst`, la condición del bucle pregunta repetidamente a la colección `files` cuántos archivos está almacenando. ¿El valor devuelto por `size` varía entre una comprobación y la siguiente? Si la respuesta es no, entonces reescriba el método de modo que el número de archivos se determine una única vez y se almacene en una variable local antes de la ejecución del bucle. Después, utilice en la condición del bucle la variable local, en lugar de invocar `size`. Compruebe que esta versión proporciona los mismos resultados. Si tiene problemas para completar este ejercicio, pruebe a utilizar el depurador para ver dónde está fallando el programa.

4.11 Mejora de la estructura: la clase `Track`

Hemos visto en un par de sitios que utilizar cadenas de caracteres para almacenar todos los detalles de las canciones no resulta enteramente satisfactorio y proporciona a nuestro reproductor de música un aspecto más bien chapucero. Cualquier reproductor

comercial nos permitiría buscar canciones según el artista, el título, el álbum, el género, etc., y probablemente incluiría otros detalles adicionales, como por ejemplo el tiempo de reproducción de la canción y el número de pista. Una de las ventajas de la orientación a objetos es que nos permite diseñar clases que modelen bastante fielmente los comportamientos y estructura inherentes de las entidades del mundo real que estemos intentando representar. Esto se consigue escribiendo clases cuyos campos y métodos se correspondan con los de los atributos. Ya sabemos lo suficiente acerca de cómo escribir clases básicas con campos, constructores y métodos selectores y mutadores, de modo que podemos diseñar fácilmente una clase `Track` que disponga, por ejemplo, de campos para almacenar por separado la información del artista y del título. De esta forma, seremos capaces de interactuar con los objetos contenidos en el organizador de música de una forma que parezca más natural.

Por tanto, es el momento de dejar de almacenar los detalles de las canciones en forma de cadenas de caracteres, porque disponer de una clase `Track` separada es la forma más apropiada para representar los elementos de datos principales –pistas de música– que vamos a utilizar en el programa. Sin embargo, no vamos a ser demasiado ambiciosos. Uno de los problemas obvios que habrá que resolver es cómo obtener los elementos independientes de información que queramos almacenar en cada objeto `Track`. Una forma sería pedir al usuario que introdujera el artista, el título, el género, etc., cada vez que añada un archivo de música al organizador. Sin embargo, esto sería bastante lento y laborioso, así que para este proyecto hemos elegido un conjunto de archivos de música que tienen el artista y el título como parte del nombre de archivo, y hemos escrito una clase auxiliar para nuestra aplicación (denominada `TrackReader`) que buscará los archivos de música contenidos en cualquier carpeta concreta y utilizará sus nombres de archivo para rellenar partes de los correspondientes objetos `Track`. No vamos a preocuparnos por el momento acerca de los detalles de cómo se lleva esto a cabo. (Más adelante en el libro, explicaremos las técnicas y clases de librería utilizadas en la clase `TrackReader`.) Una implementación de este diseño está disponible en *music-organizer-v5*.

He aquí algunos de los puntos fundamentales que conviene resaltar en esta versión:

- Lo más interesante que hay que revisar son los cambios que hemos hecho en la clase `MusicOrganizer`, al pasar de almacenar objetos `String` en la colección `ArrayList` a almacenar objetos `Track` (Código 4.7). Esto afecta a la mayoría de los métodos que hemos desarrollado anteriormente.
- Al enumerar los detalles de las canciones en `listAllTracks`, solicitamos al objeto `Track` que devuelva un objeto `String` que contenga sus detalles. Esto muestra que hemos diseñado la clase `Track` para que se responsabilice ella misma de dar formato a los detalles que hay que imprimir, como por ejemplo el artista y el título. Este es un ejemplo de lo que se conoce como *diseño dirigido por responsabilidad*, que es un concepto del que hablaremos con más detalle en un capítulo posterior.
- En el método `playTrack`, ahora tenemos que extraer el nombre de archivo del objeto `Track` seleccionado, antes de pasarlo al reproductor.
- En la librería de música, hemos añadido código para leer automáticamente de la carpeta *audio* y hemos añadido también algunas instrucciones de impresión para visualizar determinada información.

Código 4.7
Utilización de `Track` en la clase `MusicOrganizer`.

```
import java.util.ArrayList;

/**
 * Una clase para almacenar detalles de pistas de audio.
 * Pueden reproducirse las pista individuales.
 * @author David J. Barnes y Michael Kölling
 * @version 2011.07.31
 */
public class MusicOrganizer
{
    // Un ArrayList para almacenar pistas de música.
    private ArrayList<Track> tracks;
    // Un reproductor para las pistas de música.
    private MusicPlayer player;

    /* Un lector que puede leer archivos de música y
     * cargarlos como pistas.
     */
    private TrackReader reader;

    /**
     * Crear un MusicOrganizer
     */
    public MusicOrganizer()
    {
        tracks = new ArrayList<Track>();
        player = new MusicPlayer();
        reader = new TrackReader();
        readLibrary("audio");
        System.out.println("Music library loaded. " +
                           getNumberOfTracks() + " tracks.");
        System.out.println();
    }

    /**
     * Añadir una pista a la colección.
     * @param track La pista que hay que añadir.
     */
    public void addTrack(Track track)
    {
        tracks.add(track);
    }

    /**
     * Mostrar una lista de todas las pistas de la colección.
     */
    public void listAllTracks()
    {
```

Código 4.7
(*continuación*)
Utilización de `Track` en la clase `MusicOrganizer`.

```
        System.out.println("Track listing: ");
        for(Track track : tracks) {
            System.out.println(track.getDetails());
        }
        System.out.println();
    }

    /**
     * Reproducir una pista de la colección.
     * @param index El índice de la pista que hay que reproducir.
     */
    public void playTrack(int index)
    {
        if(indexValid(index)) {
            Track track = tracks.get(index);
            player.startPlaying(track.getFilename());
            System.out.println("Now playing: " + track.getArtist() +
                                      " - " + track.getTitle());
        }
    }

    Se omiten los restantes métodos.
}
```

Aunque podemos ver que el introducir una clase `Track` ha complicado ligeramente algunos de los métodos anteriores, el trabajar con objetos `Track` especializados termina proporcionando una estructura global del programa mucho mejor y nos permite desarrollar la clase `Track` hasta un nivel de detalle más apropiado para representar más información que simplemente los nombres de los archivos de música.

Con una mejor estructuración de la información de las canciones, podemos ahora proporcionar más funcionalidad a la hora de buscar canciones que satisfagan criterios concretos. Por ejemplo, si queremos localizar todas las canciones que contengan en su título la palabra *love*, podemos hacerlo de la forma siguiente, sin temor a que localicemos canciones donde esa palabra aparezca en el nombre del artista:

```
/**
 * Enumerar todas las pistas que contengan la cadena de búsqueda.
 * @param searchString La cadena de búsqueda que hay que encontrar.
 */
public void findInTitle(String searchString)
{
    for(Track track : tracks) {
        String title = track.getTitle();
        if(title.contains(searchString)) {
            System.out.println(track.getDetails());
        }
    }
}
```

Ejercicio 4.35 Añada un campo `playCount` a la clase `Track`. Proporcione métodos para reinicializar el contador a cero y para incrementarlo en una unidad.

Ejercicio 4.36 Haga que el objeto `MusicOrganizer` incremente el contador de reproducciones de una pista cada vez que se la reproduzca.

Ejercicio 4.37 Añada un campo adicional de su elección a la clase `Track` y proporcione sendos métodos, selector y mutador, para consultarlo y manipularlo. Encuentre una forma de emplear esa información en su versión del proyecto; por ejemplo, inclúyalo en la cadena de caracteres que representa los detalles de una pista, o permita configurar su valor a través de un método en la clase `MusicOrganizer`.

Ejercicio 4.38 Si reproduce dos pistas sin detener la primera, ambas se reproducirán simultáneamente. Esto no es muy útil. Modifique su programa para que la pista que se está reproduciendo se detenga automáticamente cuando se inicie otra pista distinta.

4.12 El tipo Iterator

La iteración es una herramienta vital en casi todos los proyectos de programación, así que no debería resultar sorprendente descubrir que los proyectos de programación suelen proporcionar un amplio rango de características para darle soporte, cada una con sus detalles concretos adaptados a diferentes situaciones.

Concepto:

Un **iterador** es un objeto que proporciona funcionalidad para iterar a través de todos los elementos de una colección.

Ahora vamos a presentar una tercera variante de cómo iterar a través de una colección; esta variante se encuentra en cierto modo a caballo entre el bucle while y el bucle for-each. Utiliza un bucle while para realizar la iteración y un *objeto Iterator* en lugar de una variable de índice entera para controlar la posición dentro de la lista. Tenemos que ser muy cuidadosos con la denominación en este punto, porque `Iterator` (observe la mayúscula *I*) es un *tipo* de Java, pero también nos encontraremos con un *método* denominado `iterator` (observe la minúscula *i*), así que asegúrese de prestar atención a estas diferencias al leer esta sección y al escribir su propio código.

Examinar todos los elementos de una colección es tan común, que ya hemos visto que existe un estructura de control especial –el bucle for-each– que está diseñada a propósito para esta tarea. Además, las distintas clases de librería para colecciones de Java proporcionan un tipo común diseñado a medida para soportar la iteración, y `ArrayList` es típica a este respecto.

El método `iterator` de `ArrayList` devuelve un objeto `Iterator`. `Iterator` también está definido en el paquete `java.util`, así que debemos añadir una segunda instrucción de importación al archivo de clase para poder utilizarlo:

```
import java.util.ArrayList;
import java.util.Iterator;
```

Un `Iterator` proporciona simplemente tres métodos, y dos de ellos se usan para iterar a través de una colección: `hasNext` y `next`. Ninguno de ellos admite parámetros, pero ambos tienen tipos de retorno definidos, de modo que se usan en expresiones. La forma en que normalmente utilizamos un `Iterator` puede describirse en pesudocódigo como sigue:

Iterator<*TipoElemento*> it = myCollection.iterator();
while(it.hasNext()) {
    llamar it.next() *para obtener el siguiente elemento*
    hacer algo con ese elemento
}

En este fragmento de código, utilizamos primero el método `iterator` de la clase `ArrayList` para obtener un objeto `Iterator`. Observe que `Iterator` es también un tipo genérico, por lo que lo parametrizamos con el tipo de los elementos contenidos en la colección a través de la cual estamos iterando. A continuación, empleamos ese `Iterator` para comprobar repetidamente si hay más elementos, mediante `it.hasNext()`, y para obtener el siguiente elemento, mediante `it.next()`. Un punto importante que hay que resaltar es que es al objeto `Iterator` al que le pedimos que devuelva el siguiente elemento y no al objeto colección. De hecho, tendemos a no referirnos directamente en absoluto a la colección dentro del cuerpo del bucle; toda la interacción con la colección se realiza a través del `Iterator`.

Utilizando un `Iterator`, podemos escribir un método para enumerar las pistas, como se muestra en el Código 4.8. De hecho, el `Iterator` comienza al principio de la colección y va progresando a través de ella, de objeto en objeto, cada vez que llama a su método `next`.

Código 4.8
Utilización de un `Iterator` para enumerar todas las pistas.

```
/**
 * Enumerar todas las pistas.
 */
public void listAllTracks()
{
    Iterator<Track> it = tracks.iterator();
    while(it.hasNext()) {
        Track t = it.next();
        System.out.println(t.getDetails());
    }
}
```

Tómese su tiempo para comparar esta versión con la que utiliza un bucle for-each en el Código 4.7 y con las dos versiones de `listAllFiles` mostradas en el Código 4.3 y el Código 4.5. Un aspecto concreto que hay que resaltar acerca de esta última versión es que utilizamos un bucle while, pero no necesitamos preocuparnos de la variable `index`. Esto se debe a que `Iterator` controla el punto en el que nos encontramos dentro de la colección, de modo que sabe si quedan más elementos (`hasNext`) y qué elemento devolver (`next`) si es que todavía quedan.

Una de las claves para comprender cómo funciona `Iterator` es que la llamada a `next` hace que el objeto `Iterator` devuelva el siguiente elemento de la colección *y luego avance más allá de ese elemento.* Por tanto, las llamadas sucesivas a `next` en un `Iterator` siempre devolverán elementos diferentes; no se puede volver al elemento anterior después de haber invocado `next`. En algún momento, el `Iterator` alcanzará el final de la colección y devolverá *false* al hacerse una llamada a `hasNext`. Una vez que `hasNext` ha devuelto *false*, sería un error tratar de invocar `next` sobre ese objeto `Iterator` concreto –de hecho, el objeto `Iterator` habrá sido “agotado” y ya no tendrá ninguna utilidad.

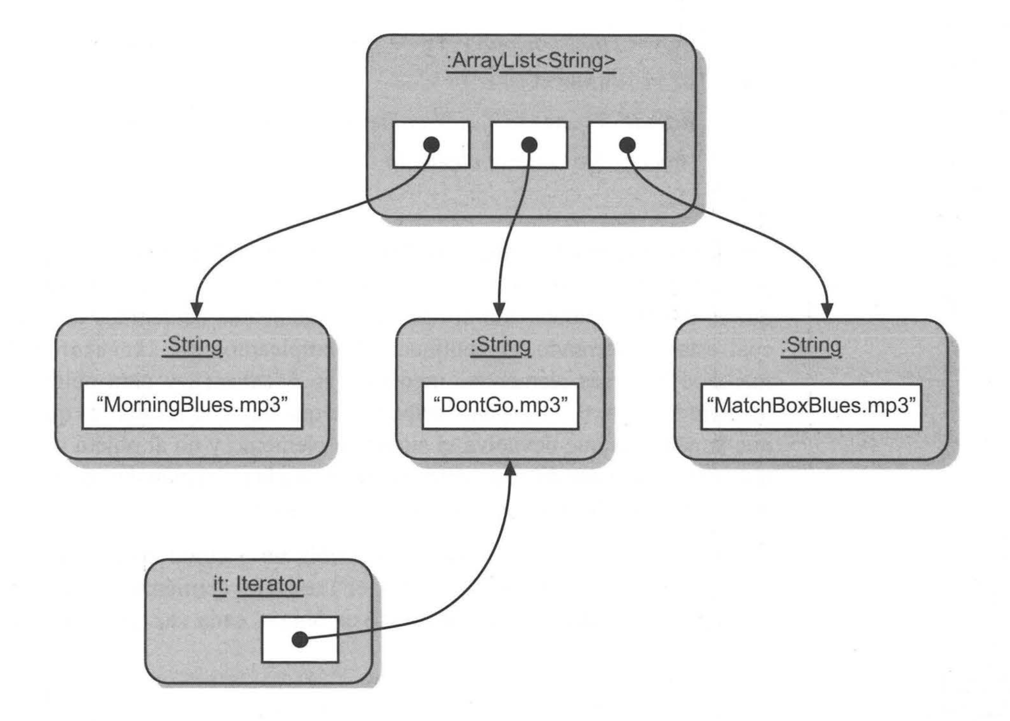


Figura 4.5
Un iterador después de una iteración apuntando al siguiente elemento que hay que procesar.

Aparentemente, `Iterator` parece no ofrecer ninguna ventaja obvia sobre las otras formas anteriores que hemos visto de iterar a través de una colección, pero las dos secciones siguientes proporcionan razones que explican por qué es importante cómo utilizarlo.

4.12.1 Comparación entre los iteradores y el acceso mediante índices

Hemos visto que tenemos al menos tres formas distintas de iterar a través de un elemento `ArrayList`. Podemos utilizar un bucle for-each (como se explica en la Sección 4.9.1), podemos usar el método `get` con una variable de índice entera (Sección 4.10.2) o un objeto `Iterator` (esta sección).

Por lo que sabemos hasta ahora, la calidad de todas estas soluciones parece aproximadamente la misma. La primera de ellas era tal vez ligeramente más fácil de comprender, pero también era la menos flexible.

La primera solución, basada en el bucle for-each, es la técnica estándar utilizada si hay que procesar todos los elementos de una colección (es decir, si estamos ante una iteración definida), ya que es la solución más concisa en ese caso. Las dos versiones siguientes presentan la ventaja de que puede detenerse la iteración en mitad del procesamiento (iteración indefinida), por lo que resultan preferibles a la hora de procesar únicamente una parte de una colección.

Para un objeto `ArrayList`, los dos últimos métodos (utilizando los bucles while) son de hecho igualmente buenos. Sin embargo, no siempre es así. Sin embargo, Java proporciona otras muchas clases de colección además de `ArrayList`; veremos algunas de ellas en los siguientes capítulos. Para algunas colecciones, es imposible o muy ineficiente acceder a los elementos individuales proporcionando un índice. Por tanto, nuestra primera versión con bucle while es una solución particular para la colección `ArrayList`, que puede no funcionar con otros tipos de colecciones.

La solución más reciente, basada en un `Iterator`, está disponible para todas las colecciones de la librería de clases Java y constituye por tanto un importante patrón de código que volveremos a utilizar en proyectos posteriores.

4.12.2 Eliminación de elementos

Otra consideración importante a la hora de seleccionar la estructura de bucle que queramos usar entra en escena cuando necesitamos poder eliminar elementos de la colección mientras estamos iterando. Un ejemplo sería desear eliminar de nuestra colección todas las canciones de un artista que ya no nos interese.

Podemos escribir esto muy fácilmente en pseudocódigo:

```
for each track en la colección {
    if track.getArtist() es el artista que ya no nos gusta:
        collection.remove(track)
}
```

Esta operación, perfectamente razonable, no se puede efectuar con un bucle for-each. Si intentamos modificar la colección utilizando uno de los métodos `remove` de la misma mientras estamos en mitad de una iteración, el sistema nos dará un error (denominado `ConcurrentModificationException`). Esto sucede porque cambiar la colección en mitad de una iteración tiene el potencial de confundir la situación enormemente. ¿Que pasa si el elemento eliminado era aquel en el que estábamos trabajando en ese momento? Si lo eliminamos, ¿cómo podemos encontrar el siguiente elemento? No existen respuestas que sean adecuadas con carácter general para estos potenciales problemas, por lo que simplemente se prohíbe la utilización del método `remove` de la colección durante una iteración del bucle for-each.

La solución apropiada para efectuar la eliminación mientras estamos iterando consiste en utilizar un `Iterator`. Su tercer método (además de `hasNext` y `next`) es `remove`. No admite ningún parámetro y tiene un tipo de retorno `void`. Invocar `remove` hará que sea eliminado el elemento devuelto por la llamada más reciente a `next`. He aquí un código de ejemplo:

```
Iterator<Track> it = tracks.iterator();
while(it.hasNext()) {
    Track t = it.next();
    String artist = t.getArtist();
    if(artist.equals(artistToRemove)) {
        it.remove();
    }
}
```

De nuevo, observe que no usamos la variable de colección `tracks` en el cuerpo del bucle. Aunque tanto `ArrayList` como `Iterator` tienen métodos `remove`, debemos utilizar el método `remove` de `Iterator`, no el de `ArrayList`.

Utilizar el método `remove` de `Iterator` es menos flexible: no podemos eliminar elementos arbitrarios, sino solo el último elemento extraído por el método `next` de `Iterator`. Por otro lado, sí que se permite utilizar el método `remove` de `Iterator` durante una iteración. Dado que el propio `Iterator` está informado de la eliminación (y se encarga de llevarla a cabo por nosotros), puede mantener apropiadamente la iteración sincronizada con la colección.

Dicha eliminación no es posible con el bucle for-each, porque no disponemos ahí de un `Iterator` con el que trabajar. En este caso, necesitamos utilizar el bucle while con un `Iterator`.

Técnicamente, también podemos eliminar elementos usando el método `get` de la colección con un índice para la iteración. Sin embargo, no le recomendamos que haga esto, porque los índices de los elementos pueden cambiar cuando añadimos o quitamos elementos y es muy fácil conseguir que la iteración opere con índices incorrectos, cuando modificamos la colección durante la iteración. El uso de un `Iterator` nos protege frente a tales errores.

Ejercicio 4.39 Implemente un método en su organizador de música que le permita especificar una cadena de caracteres como parámetro y luego elimine todas las pistas cuyos títulos contengan dicha cadena.

4.13 Resumen del proyecto *music-organizer*

En el organizador de música hemos visto cómo podemos utilizar un objeto `ArrayList`, creado a partir de una clase de la librería de clases, para almacenar un número arbitrario de objetos dentro de una colección. No tenemos que decidir de antemano cuántos objetos vamos a almacenar y el objeto `ArrayList` lleva a automáticamente la cuenta del número de elementos que almacena.

Hemos explicado cómo se puede utilizar un bucle para iterar a través de todos los elementos de la colección. Java dispone de diversas estructuras de bucle: las dos que hemos utilizado aquí son el *bucle for-each* y el *bucle while*. Normalmente, empleamos un bucle for-each cuando deseamos procesar la colección completa y el bucle while cuando no podemos saber de antemano cuántas iteraciones necesitamos o cuándo queremos poder eliminar elementos durante la iteración.

Con un `ArrayList`, podemos acceder a los elementos por su índice o iterar a través de todos los elementos mediante un objeto `Iterator`. Puede que le resulte útil repasar las diferentes circunstancias en las que son apropiados los distintos tipos de bucles (for-each y while), así como las razones por las que se prefiere emplear un `Iterator` en lugar de un índice entero, porque tendrá que tomar esta clase de decisiones una y otra vez en sus labores de programación. Tomar las decisiones correctas puede representar una gran diferencia en lo que respecta a la facilidad con la que resolver un problema concreto.

Ejercicio 4.40 Utilice el proyecto *club* para completar los siguientes ejercicios. Su tarea consistirá en completar la clase `Club`, de la que se proporciona un esbozo en el proyecto. La clase `Club` pretende almacenar dentro de una colección objetos `Membership`, que indican los miembros del club.

Dentro de `Club`, defina un campo para un `ArrayList`. Utilice una instrucción `import` apropiada para este campo y piense cuidadosamente en el tipo de elemento de la lista. En el constructor, cree el objeto colección y asígnelo al campo. Asegúrese de que todos los archivos del proyecto se compilan correctamente antes de pasar al siguiente ejercicio.

Ejercicio 4.41 Complete el método `numberOfMembers` para devolver el tamaño actual de la colección. Hasta que disponga de un método para añadir elementos a la colección, este método devolverá siempre cero, por supuesto, pero estará listo para realizar pruebas adicionales más adelante.

Ejercicio 4.42 Cada miembro de un club está representado por una instancia de la clase `Membership`. En el proyecto *club* se proporciona una versión completa de `Membership`, que no debería requerir ninguna modificación. Cada instancia contiene los detalles sobre el nombre de una persona y el mes y el año en los que se unió al club. Todos los detalles de cada miembro se rellenan al crear una instancia. Cada objeto `Membership` nuevo se añade a la colección de objetos de un `Club` mediante el método `join` del objeto `Club`, cuya descripción es la siguiente:

```
/**
 * Añadir un nuevo miembro a la colección de miembros del club.
 * @param member El objeto miembro que hay que añadir.
 */
public void join (Membership member)
```

Complete el método `join`.

Cuando desee añadir un nuevo objeto `Membership` al objeto `Club` desde el banco de objetos, hay dos formas en las que puede hacerse. Puede crear un nuevo objeto `Membership` en el banco de objetos, invocar el método `join` del objeto `Club` y hacer clic en el objeto `Membership` para suministrar el parámetro, o invocar el método `join` del objeto `Club` y escribir en el cuadro de diálogo de parámetros del método lo siguiente:

```
new Membership ("member name ...", month, year)
```

Cada vez que añada un miembro, utilice el método `numberOfMembers` para comprobar tanto que el método `join` está añadiendo los miembros a la colección, como que el método `numberOfMembers` está proporcionando el resultado correcto.

Continuaremos explorando este proyecto con otros ejercicios adicionales más adelante en el capítulo.

Ejercicio 4.43 *Ejercicio avanzado* Los siguientes ejercicios son complicados porque implican utilizar algunas cosas que no hemos tratado explícitamente. De todos modos, debería poder hacer un intento razonable, si es que comprende adecuadamente el material que hemos cubierto hasta ahora. Estos ejercicios implican añadir algo que la mayoría de los reproductores de música tienen: una característica de "reproducción aleatoria".

El paquete `java.util` contiene la clase `Random` cuyo método `nextInt` genera un entero aleatorio positivo dentro de un rango limitado. Escriba un método en la clase `MusicOrganizer` para seleccionar una única pista aleatoria de entre la lista correspondiente y reproducirla.

Sugerencia: tendrá que importar `Random` y crear un objeto `Random`, bien directamente en el método `new` o en el constructor y almacenarlo en un campo. Tendrá que localizar la documentación de la API para la clase `Random` y comprobar sus métodos para seleccionar la versión correcta de `nextInt`. De todos modos, hablaremos de la clase `Random` en el siguiente capítulo.

Ejercicio 4.44 *Ejercicio avanzado* Considere cómo podría reproducir múltiples pistas en orden aleatorio. ¿Quiere asegurarse de que todas las pistas tengan la misma oportunidad de reproducirse o prefiere sus pistas favoritas? ¿Cómo podría ayudar en este sentido un campo de "número de reproducciones" dentro la clase `Track`? Explique las diversas opciones.

Ejercicio 4.45 *Ejercicio avanzado* Escriba un método para reproducir todas las pistas de la lista de canciones exactamente una vez, pero en orden aleatorio.

Sugerencia: un forma de hacer esto sería barajar el orden de las pistas dentro de la lista –o, aun mejor dentro de una copia de la lista– y luego reproducir desde el principio hasta el final. Otra forma sería hacer una copia de la lista y luego seleccionar repetidamente una pista aleatoria de la lista, reproducirla y eliminarla de la lista, hasta que la lista esté vacía. Trate de implementar una de estas soluciones. Si prueba con la primera, ¿cómo de fácil resulta barajar la lista para que se encuentre en un orden nuevo, verdaderamente aleatorio? ¿Hay algún método de librería que pueda ayudarle en esta tarea?

4.14 Otro ejemplo: un sistema de subastas

En esta sección, vamos a profundizar en algunas de las nuevas ideas que hemos presentado en este capítulo, examinándolas en un contexto distinto.

El proyecto *auction* modela parte de la operación de un sistema de subastas en línea. La idea es que una subasta está compuesta por un conjunto de elementos que se ofrecen para la venta. Estos elementos se denominan "lotes" y el programa asigna a cada uno de ellos un número de lote distintivo. Una persona puede tratar de comprar un lote en el que esté interesada pujando una cantidad de dinero por él. Nuestras subastas son ligeramente distintas de otras existentes, porque la nuestra ofrece todos los lotes por un periodo limitado.[3] Al final de ese periodo, la subasta se cierra. Una vez cerrada la subasta, se considera que ha comprado el lote aquella persona que haya pujado una cantidad mayor por el mismo. Cualquier lote para el que no haya pujas quedará sin vender al cerrar la subasta. Los lotes no vendidos pueden ofrecerse en una subasta posterior, por ejemplo.

El proyecto *auction* contiene las siguientes clases: `Auction`, `Bid`, `Lot` y `Person`. Un examen detallado del diagrama de clases de este proyecto (Figura 4.6) revela que las relaciones entre las distintas clases son algo más complicadas que las que hemos visto en proyectos anteriores, y esto tendrá su impacto sobre la forma en la que se accede a la información durante las actividades de subasta. Por ejemplo, el diagrama muestra que los objetos `Auction`, que representan a las subastas, saben de la existencia de los otros tipos de objetos: `Bid`, `Lot` y `Person`. Los objetos `Lot`, que representan los lotes, saben de los objetos `Bid`, que representan a las pujas, y los objetos `Bid` saben acerca de los objetos `Person`, que representan a los individuos que participan en la subasta. Lo que el diagrama no nos puede decir exactamente es, por ejemplo, cómo un objeto `Auction` accede a la información almacenada en un objeto `Bid`. Para eso tenemos que examinar el código de las distintas clases.

4.14.1 Un primer vistazo al proyecto

En esta etapa, merece la pena abrir el proyecto *auction* y explorar el código fuente antes de seguir leyendo. Además de ver la familiar utilización de la clase `ArrayList` y de los bucles, probablemente se encuentre con varias cosas que resultan difíciles de comprender al principio, pero eso es algo lógico a medida que vamos introduciendo nuevas ideas y nuevas formas de hacer las cosas.

[3] En aras de la simplicidad, los aspectos de limitación temporal de las subastas no están implementados dentro de las clases que vamos a analizar aquí.

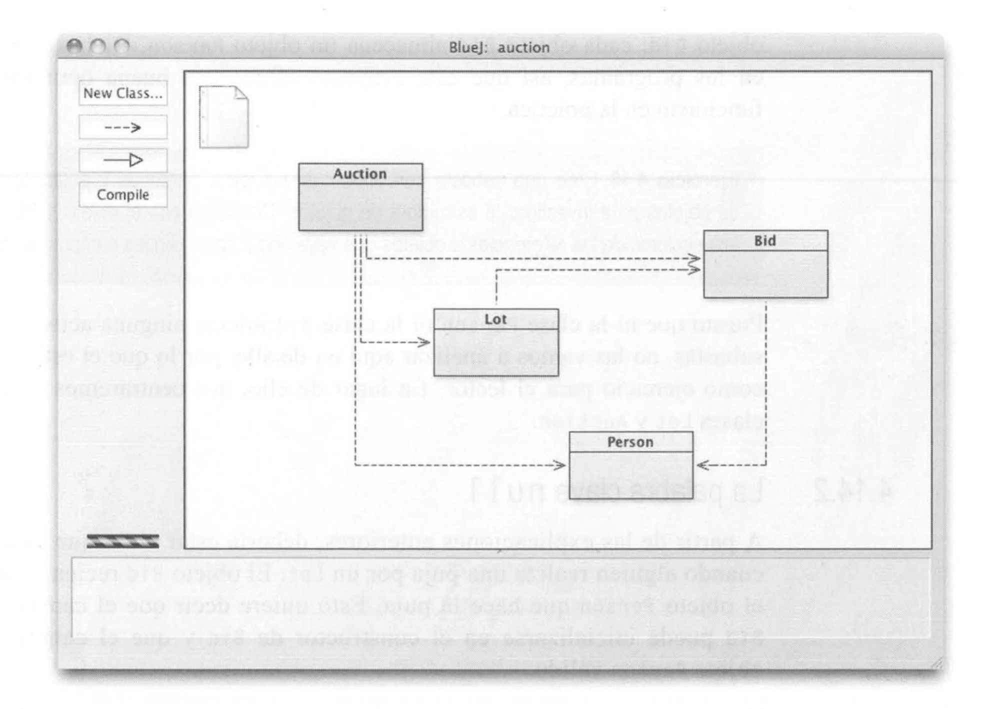


Figura 4.6
La estructura de clases del proyecto `auction`.

Un objeto `Auction` será el punto de partida para el proyecto. Las personas que quieren vender elementos los introducen en la subasta mediante el método `enterLot`, pero lo único que suministran es una cadena de caracteres con la descripción. El objeto `Auction` crea entonces un objeto `Lot` para cada lote introducido. Esto modela la forma en que funcionan las cosas en el mundo real: es la casa de subastas y no los vendedores, por ejemplo, quien asigna los números de lote o los códigos de identificación a los elementos. Por tanto, un objeto `Lot` es la representación en la casa de subastas de un elemento que está a la venta.

Para poder pujar por los lotes, las personas deben registrarse primero ante la casa de subastas. En nuestro programa, cada participante potencial en la puja está representado por un objeto `Person`. Estos objetos deben crearse independientemente en el banco de objetos de BlueJ. En nuestro proyecto, un objeto `Person` simplemente contiene el nombre de la persona. Cuando alguien quiere pujar por un lote, invoca el método `bidFor` del objeto `Auction`, introduciendo el número de lote en el que está interesado y la cantidad que está dispuesto a pagar por él. Observe que lo que se pasa es el número de lote, en lugar del objeto `Lot`; los objetos `Lot` continúan siendo internos para el objeto `Auction` y siempre se les referencia externamente mediante su número de lote.

Al igual que el objeto `Auction` crea objetos `Lot`, también transforma una puja monetaria en un objeto `Bid`, que registra la cantidad y la persona que ha efectuado la puja. Esa es la razón por la que vemos un enlace entre la clase `Bid` y la clase `Person` en el diagrama de clases. Sin embargo, observe que no existe ningún enlace entre `Bid` y `Lot`; el enlace en el diagrama va en el otro sentido, porque un objeto `Lot` almacena cuál es actualmente la puja más alta para dicho lote. Esto significa que el objeto `Lot` sustituirá el objeto `Bid` que almacena, cada vez que se realice una puja mayor.

Lo que hemos descrito pone de manifiesto una cadena bastante anidada de referencias a objetos. Los objetos `Auction` almacenan objetos `Lot`; cada objeto `Lot` puede almacenar un

objeto Bid; cada objeto Bid almacena un objeto Person. Dichas cadenas son muy comunes en los programas, así que este proyecto ofrece una buena oportunidad de explorar cómo funcionan en la práctica.

Ejercicio 4.46 Cree una subasta con unos cuantos lotes, personas y pujas. Después utilice el inspector de objetos para investigar la estructura de objetos. Comience con el objeto subasta (Auction) y continúe inspeccionando las referencias a objetos que vaya encontrando en los campos de los objetos.

Puesto que ni la clase Person ni la clase Bid inician ninguna actividad dentro del sistema de subastas, no las vamos a analizar aquí en detalle, por lo que el estudio de esas clases se deja como ejercicio para el lector. En lugar de ello, nos centraremos en el código fuente de las clases Lot y Auction.

4.14.2 La palabra clave null

A partir de las explicaciones anteriores, debería estar claro que un objeto Bid solo se crea cuando alguien realiza una puja por un Lot. El objeto Bid recién creado almacena entonces el objeto Person que hace la puja. Esto quiere decir que el campo Person de cada objeto Bid puede inicializarse en el constructor de Bid y que el campo contendrá siempre un objeto Person válido.

Por el contrario, cuando se crea un objeto Lot, esto simplemente significa que se ha introducido un lote en la subasta y ese lote no tendrá todavía ninguna puja. A pesar de ello, sigue teniendo un campo Bid, highestBid, para almacenar la puja más alta para el lote. ¿Qué valor habría que utilizar para inicializar este campo en el constructor Lot?

Concepto:

La palabra reservada null de Java se utiliza para indicar "ningún objeto" cuando una variable de objeto no está haciendo referencia actualmente a ningún objeto concreto. Todo campo que no haya sido explícitamente inicializado contendrá el valor null de manera predeterminada.

Lo que necesitamos es un valor para el campo que deje claro que todavía no hay "ningún objeto" al que esa variable esté haciendo referencia. En cierto sentido, la variable está "vacía". Para indicar esto, Java proporciona la palabra clave null. Por ello, el constructor de Lot incluye la siguiente instrucción:

```
highestBid = null;
```

Un principio muy importante es que, si una variable contiene el valor null, no se debe realizar ninguna llamada a método con ella. La razón de esto debería estar clara: como los métodos pertenecen a objetos, no podemos invocar un método si la variable no hace referencia a un objeto. Esto quiere decir que en ocasiones tenemos que usar una instrucción if para comprobar si una variable contiene null o no antes de invocar un método sobre dicha variable. Si no se hace esta comprobación, se obtendrá el error de tiempo de ejecución NullPointerException, que es muy común. Veremos algunos ejemplos de esta comprobación tanto en la clase Lot como en la clase Auction.

De hecho, si no inicializamos un campo de tipo objeto, se le dará el valor null automáticamente. Sin embargo, en este caso concreto, preferimos realizar la asignación explícitamente para que no quede ninguna duda en la mente de aquellos que lean el código de que lo que esperamos es que highestBid sea null cuando se crea un objeto Lot.

4.14.3 La clase Lot

La clase Lot almacena una descripción del lote, un número de lote y los detalles de la puja más alta recibida hasta el momento para ese lote. La parte más compleja de la clase es el

método `bidFor` (Código 4.9). Este método se encarga de lo que sucede cuando una persona realiza una puja por el lote. Cuando se hace una puja, es necesario comprobar que la nueva puja tenga un valor mayor que las demás pujas existentes para ese lote. Si es mayor, se almacenará la nueva puja dentro del lote como puja más alta actual.

Código 4.9
Gestiona una puja por un lote.

```
public class Lot
{
    // La puja más alta actual para este lote.
    private Bid highestBid;

    Se omiten otros campos y el constructor.
    /**
     * Intento de pujar por este lote. Para que tenga éxito,
     * la puja tiene que tener un valor mayor que las demás
     * pujas existentes.
     * @param bid Una nueva puja.
     * @return true si tiene éxito, false en caso contrario.
     */
    public boolean bidFor(Bid bid)
    {
        if(highestBid == null) {
            // No hay ninguna puja anterior.
            highestBid = bid;
            return true;
        }
        else if(bid.getValue() > highestBid.getValue()) {
            // La puja es más alta que la anterior.
            highestBid = bid;
            return true;
        }
        else {
            // La puja no es más alta.
            return false;
        }
    }

    Se omiten otros métodos.
}
```

Aquí, comprobamos en primer lugar si esta puja es la más alta. Eso será así si no ha habido ninguna puja anterior o si la puja es mayor que la puja más alta que haya habido hasta el momento. La primera parte de la comprobación implica asegurarse de que:

```
highestBid == null
```

Con esto se comprueba si la variable `highestBid` está actualmente haciendo referencia a un objeto o no. Como se describe en la sección anterior, hasta que se reciba una puja para este lote, el campo `highestBid` contendrá el valor `null`. Si sigue siendo `null`, entonces es que

esta es la primera puja para este lote concreto, así que obviamente será la más alta. Si no es `null`, entonces tenemos que comparar su valor con el de la nueva puja. Observe que el fallo de la primera comprobación nos proporciona una información bastante útil: ahora sabemos que `highestBid` no es `null`, así que sabemos que es seguro invocar un método sobre dicha variable. No necesitamos comprobar de nuevo el valor `null` en esta segunda condición. Comparar los valores de las dos pujas nos permite seleccionar un nueva puja más alta o rechazar la nueva puja si no es mayor que la ya existente.

4.14.4 La clase `Auction`

La clase `Auction` (Código 4.10) proporciona una ilustración adicional de la clase `ArrayList` y de los conceptos de bucles for-each que hemos presentado anteriormente en el capítulo.

Código 4.10
La clase `Auction`.

```
import java.util.ArrayList;
/**
 * Un modelo simple de una subasta.
 * La subasta mantiene una lista de lotes de longitud arbitraria.
 * @author David J. Barnes y Michael Kölling.
 * @version 2011.07.31
 */
public class Auction
{
    // La lista de lotes en esta subasta.
    private ArrayList<Lot> lots;
    // El número que se le asignará al siguiente lote introducido
    // en esta subasta.
    private int nextLotNumber;

    /**
     * Crear una nueva subasta.
     */
    public Auction()
    {
        lots = new ArrayList<Lot>();
        nextLotNumber = 1;
    }

    /**
     * Introducir un nuevo lote en la subasta.
     * @param description Una descripción del lote.
     */
    public void enterLot(String description)
    {
        lots.add(new Lot(nextLotNumber, description));
        nextLotNumber++;
    }
```

Código 4.10
(continuación)
La clase `Auction`.

```
    /**
     * Muestra la lista completa de lotes en esta subasta.
     */
    public void showLots()
    {
        for(Lot lot : lots) {
            System.out.println(lot.toString());
        }
    }

    /**
     * Hacer una puja por un lote.
     * Imprimir un mensaje indicando si la puja
     * ha tenido éxito o no.
     * @param lotNumber El lote por el que se está pujando.
     * @param bidder La persona que está pujando por el lote.
     * @param value El valor de la puja.
     */
    public void makeABid(int lotNumber, Person bidder, long value)
    {
        Lot selectedLot = getLot(lotNumber);
        if(selectedLot != null) {
            Bid bid = new Bid(bidder, value);
            boolean successful = selectedLot.bidFor(bid);
            if(successful) {
                System.out.println("The bid for lot number " +
                                   lotNumber + " was successful.");
            }
            else {
                // Indicar qué puja es mayor.
                Bid highestBid = selectedLot.getHighestBid();
                System.out.println("Lot number: " + lotNumber +
                                   " already has a bid of: " +
                                   highestBid.getValue());
            }
        }
    }

    /**
     * Devolver el lote con el número indicado. Devolver null
     * si no existe ningún lote con este número.
     * @param lotNumber El número del lote que hay que devolver.
     */
    public Lot getLot(int lotNumber)
    {
        if((lotNumber >= 1) && (lotNumber < nextLotNumber)) {
            // El número parece ser razonable.
            Lot selectedLot = lots.get(lotNumber - 1);
```

Código 4.10
(continuación)
La clase `Auction`.

```
                // Incluir una comprobación
                // para asegurarnos de tener
                // el lote correcto.
                if(selectedLot.getNumber() != lotNumber) {
                    System.out.println("Internal error: Lot number " +
                                       selectedLot.getNumber() +
                                       " was returned instead of " +
                                       lotNumber);
                    selectedLot = null;
                }
                return selectedLot;
            }
            else {
                System.out.println("Lot number: " + lotNumber +
                                   " does not exist.");
                return null;
            }
        }
}
```

El campo `lots` es un `ArrayList` utilizado para almacenar los lotes ofrecidos en esta subasta. Los lotes se introducen en la subasta pasando una descripción simple al método `enterLot`. Cada nuevo lote se crea pasando la descripción y un número de lote distintivo al constructor de `Lot`. El nuevo objeto `Lot` se añade a la colección. La siguiente sección explica algunas características adicionales bastante comunes, que se ilustran en la clase `Auction`.

4.14.5 Objetos anónimos

El método `enterLot` de `Auction` ilustra un concepto bastante común: los objetos anónimos. Podemos ver esto en la siguiente instrucción:

```
lots.add(new Lot(nextLotNumber, description));
```

Aquí estamos haciendo dos cosas:

- Estamos creando un nuevo objeto `Lot`.
- También estamos pasando este nuevo objeto al método `add` de `ArrayList`.

Podríamos haber escrito la misma instrucción en dos líneas, con el fin de que fuera más explícita la separación entre los dos pasos:

```
Lot furtherLot = new Lot(nextLotNumber, description);
lots.add(furtherLot);
```

Ambas versiones son equivalentes, pero si no vamos a dar ningún uso posterior a la variable `furtherLot`, entonces la versión original evita definir una variable con un uso tan limitado. De hecho, lo que hacemos es crear un objeto anónimo, un objeto sin nombre, y pasárselo directamente al método que va a utilizarlo.

Ejercicio 4.47 El método `makeABid` incluye las dos instrucciones siguientes:

```
Bid bid = new Bid(bidder, value);
boolean successful = selectedLot.bidFor(bid);
```

La variable `bid` solo se utiliza aquí como almacén temporal del objeto `Bid` recién creado, antes de pasarlo inmediatamente al método `bidFor` del lote. Escriba de nuevo estas instrucciones para eliminar la variable `bid` utilizando un objeto anónimo, como hemos visto en el método `enterLot`.

4.14.6 Encadenamiento de llamadas a métodos

En la introducción al proyecto *auction*, nos hemos fijado en una cadena de referencias a objetos: los objetos `Auction` almacenan objetos `Lot`; cada objeto `Lot` puede almacenar un objeto `Bid`; cada objeto `Bid` almacena un objeto `Person`. Si el objeto `Auction` necesita identificar quién tiene actualmente la puja más alta para un objeto `Lot`, entonces tendrá que pedir a `Lot` que devuelva el objeto `Bid` correspondiente a dicho lote y luego preguntar al objeto `Bid` quién es el objeto `Person` que ha efectuado la puja.

Ignorando la posibilidad de referencias `null` a objetos, podríamos ver algo similar a la siguiente secuencia de instrucciones si quisiéramos imprimir el nombre de la persona que ha hecho una puja:

```
Bid bid = lot.getHighestBid();
Person bidder = bid.getBidder();
String name = bidder.getName();
System.out.println(name);
```

Dado que las variables `bid`, `bidder` y `name` se están utilizando aquí simplemente como pasos transitorios para acceder al nombre de quien ha realizado la puja, es habitual comprimir secuencias como estas, usando referencias a objetos anónimos. Por ejemplo, podemos conseguir el mismo efecto con la instrucción siguiente:

```
System.out.println(lot.getHighestBid().getBidder().getName());
```

Esta instrucción parece sugerir que hay métodos llamando a otros métodos, pero no es así como hay que leer la instrucción. Teniendo presente que los dos conjuntos de instrucciones son equivalentes, la cadena de llamadas a métodos debe leerse estrictamente de izquierda a derecha:

```
lot.getHighestBid().getBidder().getName()
```

La llamada a `getHighestBid` devuelve un objeto `Bid` anónimo, y a continuación se invoca el método `getBidder` sobre dicho objeto. De forma similar, `getBidder` devuelve un objeto `Person` anónimo, con lo que se invoca `getName` sobre dicha persona.

Estas cadenas de llamadas a métodos pueden parecer complicadas, pero se las puede entender si se comprenden las reglas subyacentes. Incluso si decide no escribir su propio código de esta forma más concisa, sí que debe aprender a leerlo, porque puede encontrárselo en el código escrito por algún otro programador.

Ejercicio 4.48 Añada un método `close` a la clase `Auction`. Este método debe iterar a través de la colección de lotes e imprimir los detalles de todos los lotes. Utilice un bucle for-each. Cualquier lote que tenga al menos una puja se considerará vendido, de modo que lo que estamos buscando son objetos `Lot` cuyo campo `highestBid` no sea `null`. Utilice una variable local dentro del bucle para almacenar el valor devuelto por las llamadas al método `getHighestBid`, y luego compruebe si dicha variable tiene el valor `null`.

Para lotes que tengan asignada una puja, los detalles deben incluir el nombre de la persona que ha hecho la puja y el valor de esa puja más alta. Para los lotes por los que nadie haya pujado, imprima un mensaje que lo indique.

Ejercicio 4.49 Añada un método `getUnsold` a la clase `Auction` con la siguiente cabecera:

```
public ArrayList<Lot> getUnsold()
```

Este método debe iterar a través del campo `lots`, almacenando los lotes no vendidos en una nueva variable local `ArrayList`. Lo que estamos buscando son los objetos `Lot` cuyo campo `highestBid` sea `null`. Al final del método, devuelva la lista de los lotes no vendidos.

Ejercicio 4.50 Suponga que la clase `Auction` incluye un método que hace posible eliminar un lote de la subasta. Suponiendo que el campo `lotNumber` de los lotes restantes no se modifica al eliminar un lote, escriba cuál cree que sería el impacto sobre el método `getLot`.

Ejercicio 4.51 Escriba de nuevo `getLot` para que no dependa del hecho de que un lote con un número concreto esté almacenado en el índice `(number-1)` de la colección. Por ejemplo, si se ha eliminado el lote número 2, entonces el lote número 3 se habrá desplazado del índice 2 al índice 1, y todos los lotes con número más alto también se habrán desplazado una posición de índice. Puede asumir que los lotes siempre se almacenan por orden creciente de sus números de lote.

Ejercicio 4.52 Añada un método `removeLot` a la clase `Auction`, que tenga la siguiente cabecera:

```
/**
 * Eliminar el lote con el número de
 * lote especificado.
 * @param number El número del lote que hay que eliminar.
 * @return El lote con el número dado o null si
 *         no existe tal lote.
 */
public Lot removeLot(int number)
```

Este método no debe suponer que un lote con un número dado está almacenado en ninguna posición concreta dentro de la colección.

Ejercicio 4.53 La clase `ArrayList` está disponible en el paquete `java.util`. Dicho paquete también incluye una clase denominada `LinkedList`. Averigüe lo que pueda acerca de la clase `LinkedList` y compare sus métodos con los de `ArrayList`. ¿Qué métodos tienen en común y cuáles son diferentes?

4.14.7 Utilización de colecciones

La clase de colección `ArrayList` (y otras como ella) constituye una herramienta de programación importante, porque muchos problemas de programación implican trabajar con

colecciones de objetos de tamaño variable. Antes de continuar con el capítulo es importante familiarizarse en profundidad con el modo de trabajar con las clases, hasta sentirse cómodo con ellas. Los siguientes ejercicios le ayudarán en este sentido.

Ejercicio 4.54 Continúe trabajando con el proyecto *club* del Ejercicio 4.40. Defina un método en la clase `Club` con la siguiente descripción:

```
/**
 * Determinar el número de miembros que se han unido
 * en el mes indicado.
 * @param month El mes que nos interesa.
 * @return El número de miembros que se han unido en ese mes.
 */
public int joinedInMonth(int month)
```

Si el parámetro `month` está fuera del rango válido de 1 a 12, imprimir un mensaje de error y devolver cero.

Ejercicio 4.55 Defina un método en la clase `Club` con la siguiente descripción:

```
/**
 * Eliminar de la colección del club todos los miembros
 * que se hayan unido en el mes especificado y devolverlos
 * almacenados en un objeto colección separado.
 *
 * @param month El mes de ingreso en el club.
 * @param year El año de ingreso en el club.
 * @return Los miembros que se han unido al club en el mes
 *         y año indicados.
 */
public ArrayList<Membership> purge(int month, int year)
```

Si el parámetro `month` está fuera del rango válido de 1 a 12, imprimir un mensaje de error y devolver un objeto colección que no tenga ningún objeto almacenado.

Nota: El método `purge` es significativamente más difícil de escribir que cualquiera de los otros métodos de esta clase.

Ejercicio 4.56 Abra el proyecto *product* y complete la clase `StockManager` en este y los siguientes ejercicios. `StockManager` utiliza un `ArrayList` para almacenar elementos `Product`. Su método `addProduct` ya se encarga de añadir un producto a la colección, pero es preciso completar los siguientes métodos: `delivery`, `findProduct`, `printProductDetails` y `numberInStock`.

Cada producto vendido por la empresa está representado mediante una instancia de la clase `Product`, que almacena un ID de producto, un nombre y cuántos elementos de dicho producto hay en este momento en el almacén. La clase `Product` define el método `increaseQuantity` para registrar incrementos en la cantidad de dicho producto existente en almacén. El método `sellOne` registra que se ha vendido un elemento de dicho producto, reduciendo en una unidad el campo que refleja la cantidad. `Product` ya está escrita, y no debería necesitar efectuar ninguna modificación en la misma.

Empiece implementando el método `printProductDetails` que se encarga de imprimir los detalles del producto, para cerciorarse de que es capaz de iterar a través de la colección de productos. Imprima simplemente los detalles de cada `Product` devuelto, invocando el método `toString`.

Ejercicio 4.57 Implemente el método `findProduct`. Este método debe buscar en la colección un producto cuyo campo `id` se corresponda con el argumento ID de este método. Si se encuentra el producto correspondiente, debe devolverse como resultado del método. Si no se encuentra ningún producto, hay que devolver `null`.

Este método difiere del método `printProductDetails` en que no será necesario examinar cada producto de la colección antes de encontrar una correspondencia. Por ejemplo, si el primer producto de la colección se corresponde con el ID del producto, la iteración puede terminar y puede devolverse ese primer objeto `Product`. Por otro lado, es posible que no se encuentre ninguna correspondencia dentro de la colección. En dicho caso, se examinará toda la colección sin encontrar un producto que el método pueda devolver. En este caso, debe devolverse el valor `null`.

Al buscar una correspondencia, tendrá que invocar el método `getID` sobre un objeto `Product`.

Ejercicio 4.58 Implemente el método `numberInStock`. Este método debe localizar un producto dentro de la colección que tenga un ID que se corresponda con el suministrado, y devolver como resultado del método la cantidad actualmente existente de dicho producto. Si no se encuentra ningún producto que se corresponda con el ID suministrado, devuelve cero. Este método es relativamente simple de implementar una vez que se ha completado el método `findProduct`. Por ejemplo, `numberInStock` puede llamar al método `findProduct` para realizar la búsqueda y llamar al método `getQuantity` con el resultado del método anterior. Tenga cuidado, sin embargo, con los productos que no se puedan encontrar.

Ejercicio 4.59 Implemente el método `delivery`, que indica una recepción de productos, utilizando un enfoque similar al empleado en `numberInStock`. Debe localizar en la lista de productos el producto que tenga el ID especificado y luego llamar a su método `increaseQuantity`.

Ejercicio 4.60 *Ejercicio avanzado* Implemente un método `StockManager` para imprimir detalles de todos los productos cuya cantidad en almacén esté por debajo de un valor dado (que se pasa como parámetro al método).

Modifique el método `addProduct` para que no pueda añadirse a la lista un nuevo producto que tenga el mismo ID que un producto ya existente.

Añada un método a `StockManager` que localice un producto a partir de su nombre en lugar de a partir de su ID.

```
public Product findProduct(String name)
```

Para hacer esto, necesitará saber que se puede comprobar la igualdad entres dos objetos `String`, `s1` y `s2`, utilizando la expresión booleana

```
s1.equals(s2)
```

Puede encontrar más detalles acerca de esto en el Capítulo 5.

4.15 Resumen sobre colecciones flexibles

Hemos visto que las clases como `ArrayList` nos permiten crear cómodamente colecciones que contengan un número arbitrario de objetos. La librería Java contiene más colecciones como esta, y examinaremos algunas de ellas en el siguiente capítulo. Comprobará que ser capaz de utilizar las colecciones constituye una habilidad importante a la hora de escribir programas interesantes. Apenas existen aplicaciones, entre las que veremos a partir de ahora, que no utilicen colecciones de una forma u otra.

4.16 Colecciones de tamaño fijo

Las colecciones de tamaño flexible son muy potentes porque no necesitamos saber de antemano cuántos elementos vamos a almacenar en ellas y porque se puede variar el número de elementos que almacenan. Sin embargo, algunas aplicaciones son diferentes, en el sentido de que *sí que sabemos* de antemano cuántos elementos vamos a querer almacenar en una colección, y ese número suele permanecer fijo durante todo el tiempo de vida de la colección. En estas circunstancias, tenemos la opción de seleccionar un objeto colección especializado, de tamaño fijo, para almacenar los elementos.

Concepto:

Una **matriz** es un tipo especial de colección que puede almacenar un número fijo de elementos.

Una colección de tamaño fijo se denomina *matriz.* Aunque el tamaño fijo de las matrices puede ser una desventaja significativa en muchas situaciones, tienen a cambio dos ventajas frente a las clases de colección de tamaño flexible:

- El acceso a los elementos almacenados en una matriz suele ser más eficiente que el acceso a los elementos de una colección de tamaño flexible comparable.
- Las matrices pueden almacenar tanto objetos como valores de tipo primitivo. Las colecciones de tamaño flexible solo pueden almacenar objetos.[4]

Otra característica distintiva de las matrices es que tiene un soporte sintáctico especial en Java; se puede acceder a ellas utilizando una sintaxis personalizada que difiere de las llamadas tradicionales a métodos. La razón de esto es principalmente histórica: las matrices son la estructura de colección más antigua utilizada en los lenguajes de programación, y la sintaxis para tratar con matrices se ha ido desarrollando a lo largo de muchas décadas. Java utiliza la misma sintaxis establecida en otros lenguajes de programación para hacer más sencillas las cosas para aquellos programadores que ya estén empleando matrices, aun cuando este tratamiento no sea coherente con el resto de la sintaxis del lenguaje.

En las siguientes secciones, mostraremos cómo pueden utilizarse las matrices para mantener colecciones de tamaño fijo. También introduciremos una nueva estructura de bucle que está estrechamente asociada muy a menudo con las matrices –el *bucle for.* (Observe que el *bucle for* es diferente del *bucle for-each.*)

---

[4] Una estructura Java denominada *autoboxing* proporciona un mecanismo que también nos permite almacenar valores primitivos en colecciones de tamaño flexible. Sin embargo, es cierto que solo las matrices permiten almacenarlos de forma directa.

4.16.1 Un analizador de archivo de registro

Los servidores web suelen mantener archivos de registro de los accesos de los clientes a la páginas web que almacenan. Con las herramientas adecuadas, estos registros permiten a los administradores de servicios web extraer y analizar información útil, como por ejemplo:

- cuáles son las páginas más populares que proporcionan,
- qué sitios han derivado usuarios hacia este sitio,
- si otros sitios parecen tener vínculos rotos a las páginas de este sitio,
- cuántos datos se están transmitiendo a los clientes,
- los periodos de mayor uso a lo largo de un día, una semana o un mes.

Dicha información puede ayudar a los administradores a determinar, por ejemplo, si necesitan actualizarse y emplear máquinas servidoras más potentes o bien determinar cuáles son los periodos de menor actividad con el fin de planificar las actividades de mantenimiento.

El proyecto *weblog-analyzer* contiene una aplicación que realiza un análisis de los datos de uno de esos servidores web. El servidor escribe una línea en un archivo de registro cada vez que se produce un acceso. En la carpeta del proyecto se proporciona un archivo de registro de ejemplo denominado *weblog.txt*. Cada línea registra el día y la hora del acceso en el siguiente formato:

```
año mes día hora minutos
```

Por ejemplo, la siguiente línea registra un acceso a las 03:45 de la madrugada del 7 de junio de 2011:

```
2011 06 07 03 45
```

El proyecto está compuesto por cinco clases: `LogAnalyzer`, `LogfileReader`, `LogEntry`, `LoglineTokenizer` y `LogfileCreator`. Invertiremos casi todo nuestro esfuerzo en analizar la clase `LogAnalyzer`, ya que contiene ejemplos de creación y utilización de una matriz (Código 4.11). En ejercicios posteriores se le pedirá que examine y modifique `LogEntry`, porque también emplea una matriz. Las clases `LogReader` y `LogLineTokenizer` utilizan características del lenguaje Java que todavía no hemos visto, por lo que no vamos a explorarlas en detalle. La clase `LogfileCreator` permite crear nuestros propios archivos de registro con datos aleatorios.

Código 4.11
El analizador de archivos de registro.

```
/**
 * Leer los datos del servidor web y analizar
 * los patrones de acceso horario.
 *
 * @author David J. Barnes y Michael Kölling.
 * @version 2011.07.31
 */
public class LogAnalyzer
{
    // Matriz para almacenar el número de accesos en cada hora.
    private int[] hourCounts;
```

Código 4.11
(*continuación*)
El analizador de archivos de registro.

```
    // Usa un LogfileReader para acceder a los datos.
    private LogfileReader reader;

    /**
     * Crea un objeto para analizar los accesos web que se
     * han producido en cada hora.
     */
    public LogAnalyzer()
    {
        // Crea el objeto matriz para almacenar el número
        // de accesos producidos en cada hora.
        hourCounts = new int[24];
        // Crea el lector para obtener los datos.
        reader = new LogfileReader();
    }

    /**
     * Analiza los datos de acceso horario a partir
     * del archivo de registro.
     */
    public void analyzeHourlyData()
    {
        while(reader.hasNext()) {
            LogEntry entry = reader.next();
            int hour = entry.getHour();
            hourCounts[hour]++;
        }
    }

    /**
     * Imprimir el número de accesos en cada hora.
     * Estos datos deben haberse obtenido mediante una llamada
     * previa a analyzeHourlyData.
     */
    public void printHourlyCounts()
    {
        System.out.println("Hr: Count");
        for(int hour = 0; hour < hourCounts.length; hour++) {
            System.out.println(hour + ": " + hourCounts[hour]);
        }
    }

    /**
     * Imprimir las líneas de datos leídas por LogfileReader
     */
    public void printData()
    {
        reader.printData();
    }
}
```

Actualmente el analizador utiliza solo una parte de los datos almacenados en la línea de registro de un servidor. Proporciona información que nos permitiría determinar qué horas del día tienden a ser, como promedio, las de mayor y menor carga para el servidor. Hace esto contando cuántos accesos se produjeron en cada periodo de una hora, a lo largo del tiempo cubierto por el archivo de registro.

Ejercicio 4.61 Explore el proyecto *weblog-analyzer* creando un objeto `LogAnalyzer` e invocando su método `analyzeHourlyData`. Después, llame al método `printHourlyCounts`, que imprimirá los resultados del análisis. ¿Cuáles son los momentos de día con mayor carga?

A lo largo de las siguientes secciones examinaremos la forma en que esta clase utiliza una matriz para llevar a cabo su tarea.

4.16.2 Declaración de variables de matriz

La clase `LogAnalyzer` contiene un campo que es de tipo matriz:

```
private int[] hourCounts;
```

La característica distintiva de la declaración de una variable de tipo matriz es una pareja de corchetes que forman parte del nombre del tipo: `int[]`. Esto indica que la variable `hourCounts` es del tipo *matriz de enteros*. Decimos en este caso que `int` es el *tipo base* de esta matriz concreta, lo que significa que el objeto matriz almacenará valores de tipo `int`. Es importante distinguir entre la declaración de una variable de matriz y la declaración de una variable simple que tiene un aspecto similar:

```
int hour; // Un única variable int.
int[] hourCounts; // Una variable de matriz de int.
```

Aquí, la variable `hour` es capaz de almacenar un único valor entero, mientras que `hourCounts` se utilizará para referirnos a un objeto matriz, después de haber creado dicho objeto. La declaración de una variable de matriz no crea por sí misma el objeto matriz. Eso se lleva a cabo en un etapa separada, utilizando el operador `new`, como se hace con otros objetos.

Merece la pena examinar de nuevo esta sintaxis tan inusual. La declaración `int[]` podría aparecer, quizá, como `Array<int>` en una sintaxis más convencional. El que no sea así obedece a razones históricas más que lógicas. De todos modos, acostúmbrese a leer la declaración de la misma forma: como una "matriz (*array*) de int."

Ejercicio 4.62 Escriba una declaración para una variable matriz `people` que pudiera utilizarse para hacer referencia a una matriz de objetos `Person`.

Ejercicio 4.63 Escriba una declaración para una variable matriz `vacant` que pudiera utilizarse para hacer referencia a una matriz de valores booleanos.

Ejercicio 4.64 Lea la clase `LogAnalyzer` e identifique todos los lugares en los que se utiliza la variable `hourCounts`. Por el momento, no se preocupe por lo que significan todos esos usos, ya que los explicaremos en las siguientes secciones. Observe en cuántas ocasiones se emplea una pareja de corchetes con la variable.

Ejercicio 4.65 ¿Qué error hay en las siguientes declaraciones de matrices? Corríjalos.

```
[]int counts;                    boolean[5000] occupied;
```

4.16.3 Creación de objetos matriz

Lo siguiente que tenemos que ver es cómo se asocia una variable matriz con un objeto matriz.

El constructor de la clase `LogAnalyzer` incluye una instrucción para crear un objeto matriz `int`:

```
hourCounts = new int[24];
```

De nuevo, observe lo diferente que es la sintaxis con respecto a la de la creación de objetos normales. Por ejemplo, no hay paréntesis para los parámetros de un constructor, porque un objeto matriz no tiene constructor. Esta instrucción crea un objeto matriz que es capaz de almacenar 24 valores enteros independientes y hace que la variable matriz `hourCounts` haga referencia a dicho objeto. El valor 24 es el tamaño de la matriz y no un parámetro de ningún constructor. La Figura 4.7 ilustra el resultado de esta asignación.

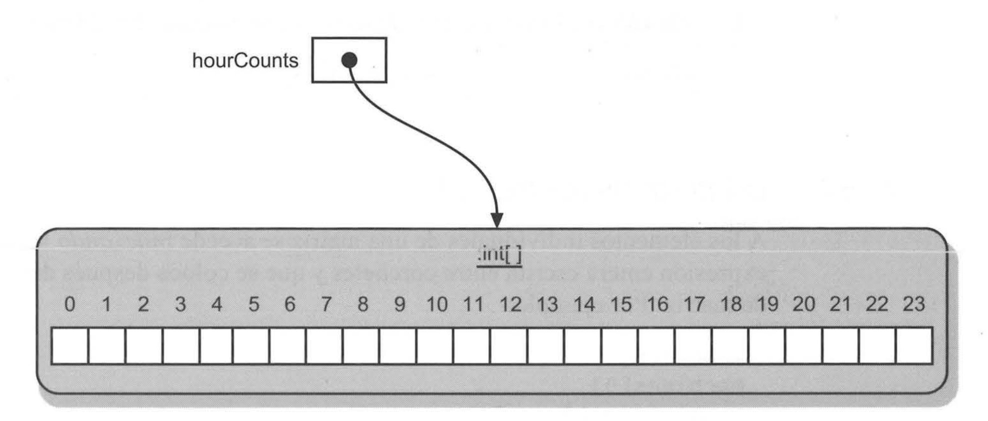


Figura 4.7
Una matriz de 24 enteros.

La forma general de construcción de un objeto matriz es:

`new` *tipo*`[`*expresión-entera*`]`

La elección de *tipo* especifica el tipo de elemento que se almacenará en la matriz. La *expresión-entera* especifica el tamaño de la matriz; es decir, el número fijo de elementos que puede almacenarse en ella.

Cuando se asigna un objeto matriz a una variable matriz, el tipo del objeto matriz debe corresponderse con el tipo declarado para la variable. La asignación a `hourCounts` se permite porque el objeto matriz es una matriz de enteros y `hourCounts` es una variable matriz de enteros. La siguiente instrucción declara una variable de matriz de cadenas y hace referencia a una matriz que tiene una capacidad de almacenamiento de 10 cadenas de caracteres:

```
String[] names = new String[10];
```

Es importante observar que la creación de la matriz asignada a `names` no crea en la práctica 10 cadenas de caracteres. Más bien, lo que hace es crear una colección de tamaño fijo que es capaz de almacenar 10 cadenas dentro de ella. Esas cadenas probablemente se creen en otra parte de la clase a la que pertenece `names`. Inmediatamente después de su creación, podemos considerar que el objeto matriz está vacío. Si es una matriz para objetos, entonces contendrá valores `null` en todos sus elementos. Si es una matriz `int`, entonces todos los elementos se inicializarán con el valor cero. En la siguiente sección examinaremos la forma en que se almacenan los elementos en las matrices y en la que se extraen de las mismas.

Ejercicio 4.66 Dadas las siguientes declaraciones de variables,

```
double[] readings;
String[] urls;
TicketMachine[] machines;
```

escriba asignaciones que lleven a cabo las siguientes tareas: (a) hacer que la variable `readings` haga referencia a una matriz capaz de almacenar sesenta valores `double`; (b) hacer que la variable `urls` haga referencia a una matriz capaz de almacenar noventa objetos `String`; (c) hacer que la variable `machines` haga referencia a una matriz capaz de almacenar cinco objetos `TicketMachine`.

Ejercicio 4.67 ¿Cuántos objetos `String` se crean en la siguiente declaración?

```
String[] labels = new String[20];
```

Ejercicio 4.68 ¿Cuál es el error en la siguiente creación de una matriz? Corríjalo.

```
double[] prices = new double(50);
```

4.16.4 Utilización de objetos matriz

A los elementos individuales de una matriz se accede *indexando* la matriz. Un índice es una expresión entera escrita entre corchetes y que se coloca después del nombre de una variable de matriz. Por ejemplo:

```
labels[6]
machines[0]
people[x + 10 - y]
```

Los valores válidos para una expresión de índice dependen de la longitud de la matriz con la que se esté trabajando. Al igual que sucede con otras colecciones, los índices de una matriz siempre comienzan en cero y van hasta una unidad menos que la longitud de la matriz. Por tanto, los índices válidos para la matriz `hourCounts` van de 0 a 23, ambos inclusive.

Error común Dos errores muy comunes son pensar que los índices válidos de una matriz comienzan en 1 y utilizar el valor de la longitud de la matriz como índice. El uso de índices fuera de los límites legales de una matriz hará que se produzca un error de tiempo de ejecución denominado `ArrayIndexOutOfBoundsException`.

Las expresiones que seleccionan un elemento de una matriz pueden utilizarse en cualquier lugar donde se pueda emplear una variable del tipo base de la matriz. Esto significa que podemos, por ejemplo, utilizar esas expresiones en ambos lados de una asignación. He aquí algunos ejemplos en los que se usan expresiones de matriz en diferentes lugares:

```
labels[5] = "Quit";
double half = readings[0] / 2;
System.out.println(people[3].getName());
machines[0] = new TicketMachine(500);
```

La utilización de un índice de matriz en el lado izquierdo de una instrucción de asignación es el equivalente, dentro de las matrices, a un método mutador (o método `set`), porque se modifi-

cará el contenido de la matriz. La utilización de un índice de matriz en cualquier otro lugar es el equivalente de un método selector (o método get).

4.16.5 Análisis del archivo de registro

La matriz hourCounts creada en el constructor de LogAnalyzer se utiliza para almacenar un análisis de los datos de acceso. Los datos se almacenan en esa matriz en el método analyzeHourlyData y se extraen de ella para visualizarlos en el método printHourlyCounts. Como la tarea del método analyze es contar cuántos accesos se han producido durante cada periodo de una hora, la matriz necesita 24 posiciones –una para cada periodo horario de un día de 24 horas. El analizador delega la tarea de leer su archivo de registro a un LogfileReader.

La clase LogfileReader es bastante compleja y le sugiero que no dedique demasiado tiempo a investigar su implementación. Su función consiste en gestionar la tarea de descomponer cada línea del registro en valores de datos separados, pero podemos abstraernos de los detalles de implementación considerando simplemente las cabeceras de dos de sus métodos:

```
public boolean hasNext()
public LogEntry next()
```

Estos métodos se corresponden exactamente con los métodos que hemos visto con el tipo Iterator y un LogfileReader puede emplearse exactamente de la misma forma, salvo porque no permitimos utilizar el método remove. El método hasNext le dice al analizador si existe al menos una entrada más en el archivo de registro, mientras que el método next devuelve un objeto LogEntry que contiene los valores de la siguiente línea del registro.

A partir de cada LogEntry, el método analyzeHourlyData del analizador obtiene el valor del campo de hora:

```
int hour = entry.getHour();
```

Sabemos que el valor almacenado en la variable local hour siempre estará comprendido en el rango que va de 0 a 23, lo que se corresponde exactamente con el rango de índices válidos para la matriz hourCounts. Cada posición de la matiz se utiliza para representar el número de accesos que se han producido en la hora correspondiente. Por tanto, cada vez que se lee un valor de hora, lo que queremos es actualizar la cuenta para dicha hora incrementándola en 1. Hemos escrito esto de la forma siguiente:

```
hourCounts[hour]++;
```

Observe que lo que se incrementa es el valor almacenado en el elemento de la matriz y no la variable hour. Las siguientes alternativas son también equivalentes, ya que podemos utilizar un elemento de una matriz exactamente de la misma forma que emplearíamos una variable ordinaria:

```
hourCounts[hour] = hourCounts[hour] + 1;
hourCounts[hour] += 1;
```

Al finalizar el método analyzeHourlyData, dispondremos de un conjunto completo del número acumulado de accesos para cada hora del periodo de registro.

En la siguiente sección, echaremos un vistazo al método printHourlyCounts, ya que introduce una nueva estructura de control que está muy bien adaptada para iterar a través de una matriz.

4.16.6 El bucle for

Java define dos variantes de bucles for, indicándose ambas mediante la palabra clave *for* en el código fuente. En la Sección 4.9, hemos presentado la primera de esas variantes, el *bucle for-each*, que es un método conveniente para iterar a través de una colección de tamaño flexible. La segunda variante, el *bucle for*, es una estructura de control iterativo alternativa[5], que es particularmente apropiada cuando:

- queremos ejecutar un cierto conjunto de instrucciones un número fijo de veces,
- necesitamos una variable dentro del bucle cuyo valor cambie en una cantidad fija –incrementándose normalmente en 1– en cada iteración.

El bucle for está bien adaptado a aquellas situaciones en las que se necesita una *iteración definida.* Por ejemplo, es común emplear un bucle for cuando queremos hacer algo con todos los elementos de una matriz, como por ejemplo imprimir el contenido de cada elemento. Esto encaja con el criterio, ya que el número fijo de veces se corresponde con la longitud de la matriz y nos hace falta una variable para proporcionar un índice incremental para la matriz.

Un bucle for tiene la siguiente forma general:

```
for(inicialización; condición; acción post-cuerpo) {
    instrucciones que hay que repetir
}
```

El siguiente ejemplo está extraído del método `printHourlyCounts` del `LogAnalyzer`:

```
for(int hour = 0; hour < hourCounts.length; hour++) {
    System.out.println(hour + ": " + hourCounts[hour]);
}
```

El resultado de esto es que se imprimirá el valor de cada elemento de la matriz, precedido por su correspondiente número de hora. Por ejemplo:

```
0: 149
1: 149
2: 148
...
23: 166
```

Cuando comparamos este bucle for con el bucle for-each, observamos que la diferencia sintáctica se encuentra en la sección situada entre los paréntesis, en la cabecera del bucle. En este bucle for, los paréntesis contienen tres secciones independientes, separadas por caracteres de punto y coma.

Desde el punto de vista del diseño de un lenguaje de programación, hubiera resultado más bonito emplear dos palabras clave distintas para estos dos bucles, quizá `for` y `foreach`. La razón de que `for` se emplee para ambas variantes es, de nuevo, un accidente histórico. Las versiones más antiguas del lenguaje Java no contenían el bucle for-each y cuando se introdujo

[5] En ocasiones, cuando alguien quiere hacer más clara la distinción entre el bucle for y el bucle for-each, suele referirse al primero de ellos como el "bucle for de estilo antiguo", porque ha formado parte del lenguaje Java durante más tiempo que el bucle for-each. El bucle for-each se denomina también por eso, en ocasiones, "bucle for mejorado".

finalmente, los diseñadores de Java no querían añadir una nueva palabra clave en esa etapa, porque eso podría provocar problemas con los programas existentes. Así que decidieron utilizar la misma palabra clave para ambos bucles. Esto hace que nos resulte ligeramente más difícil distinguir estos dos bucles, pero es cuestión de práctica el acostumbrarse a reconocer las diferentes estructuras de cabecera.

Incluso aunque el bucle for se emplea a menudo para iteración definida, el hecho de que esté controlado por una expresión booleana de carácter general indica que se aproxima más al bucle while que al bucle for-each. Podemos ilustrar la forma en que se ejecuta un bucle for escribiendo su formato general mediante un bucle while equivalente:

```
inicialización;
while(condición) {
    instrucciones que hay que repetir
    acción post-cuerpo
}
```

De modo que la forma alternativa para el cuerpo de `printHourlyCounts` sería

```
int hour = 0;
while(hour < hourCounts.length) {
    System.out.println(hour + ": " + hourCounts[hour]);
    hour++;
}
```

A partir de esta versión, podemos ver que la acción post-cuerpo no se llega a ejecutar hasta después de haberse ejecutado las instrucciones contenidas en el cuerpo del bucle, a pesar de que esa acción se defina dentro de la cabecera del bucle for. Además, podemos ver que la parte de inicialización solo se ejecuta una vez, inmediatamente antes de probar la condición por primera vez.

En ambas versiones, fíjese especialmente en la condición

```
hour < hourCounts.length
```

Esto ilustra dos puntos importantes:

- Todas las matrices tienen un campo `length` que contiene el valor de su tamaño fijo. El valor de este campo siempre se corresponderá con el valor de la expresión entera utilizada para crear el objeto matriz. Por tanto, el valor de `length` aquí será 24.
- La condición utiliza el operador menor que, <, para comparar el valor de `hour` con la longitud de la matriz. Por tanto, en este caso, el bucle continuará ejecutándose mientras que `hour` sea menor que 24. En general, cuando queremos acceder a todos los elementos de una matriz, la cabecera del bucle for tendrá el siguiente formato general:

  ```
  for(int index = 0; index < matriz.length; index++)
  ```

  Esto es correcto, porque no queremos utilizar un valor de índice que sea igual a la longitud de la matriz; dicho elemento no puede nunca existir.

4.16.7 Matrices y el bucle for-each

¿Podríamos también escribir de nuevo el bucle for mostrado anteriormente como un bucle for-each? La respuesta es: casi, casi. He aquí un intento:

```
for(int value : hourCounts) {
    System.out.println(": " + value);
}
```

Este código se compilará y ejecutará correctamente (¡Pruébelo!). A partir de este fragmento de código podemos ver que las matrices pueden, de hecho, utilizarse en bucles for-each como cualquier otra colección. Sin embargo, tenemos un problema: no podemos imprimir fácilmente la hora delante del signo de dos puntos. Este fragmento de código simplemente omite la impresión de la hora y se limita a imprimir el carácter de dos puntos y el valor. Esto se debe a que el bucle for-each no proporciona acceso a una variable que actúe como contador de bucle, y en este caso necesitamos esa variable para imprimir la hora.

Para corregir esto, necesitaríamos definir nuestra propia variable de contador (de forma similar a como hicimos en el ejemplo del bucle while). Pero, en lugar de hacer esto, preferimos emplear el bucle for de estilo antiguo, ya que es más conciso.

4.16.8 El bucle for y los iteradores

En la Sección 4.12.2, hemos mostrado la necesidad de utilizar un `Iterator` si queríamos eliminar elementos de una colección. Existe un uso especial del bucle for con un `Iterator` cuando deseamos hacer algo así. Suponga que deseáramos eliminar de nuestro organizador de música todas la canciones de un artista concreto. Lo importante aquí es que tenemos que examinar todas las canciones de la colección, por lo que un bucle for-each sería adecuado, pero ya sabemos que no podemos emplearlo en este caso concreto. Sin embargo, podemos utilizar un bucle for de la manera siguiente:

```
for(Iterator<Track> it = tracks.iterator(); it.hasNext(); ) {
    Track t = it.next();
    if(t.getArtist().equals(artist)) {
        it.remove();
    }
}
```

El punto importante aquí es que no hay ninguna acción post-cuerpo del bucle –nos hemos limitado a dejarla en blanco. Esto es perfectamente legal, pero seguimos teniendo que incluir el punto y coma después de la condición del bucle. Utilizando un bucle for en lugar de un bucle while, queda algo más claro que pretendemos examinar todos los elementos de la lista.

Ejercicio 4.69 Compruebe lo que sucede si se escribe incorrectamente la condición del bucle for, utilizando el operador <= en `printHourlyCounts`:

```
for(int hour = 0; hour <= hourCounts.length; hour++)
```

Ejercicio 4.70 Escriba de nuevo el cuerpo de `printHourlyCounts` para sustituir el bucle for por un bucle while equivalente. Invoque el método reescrito para comprobar que imprime los mismos resultados que antes.

¿Qué bucle debo utilizar? Hemos explicado tres bucles diferentes: el bucle for-each, el bucle while y el bucle for. Como hemos visto, en muchas situaciones podemos elegir cualquiera de esos tipos de bucle para resolver nuestra tareas. Normalmente, un bucle puede reescribirse utilizando otro tipo de bucle. Teniendo en cuenta todo esto, ¿cómo decidimos qué bucle utilizar en cualquier punto concreto? He aquí algunas directrices:

- Si necesitamos iterar a través de todos los elementos de una colección, el bucle for-each es casi siempre la solución más elegante. Es claro y conciso, pero no nos proporciona un contador de bucle.
- Si tenemos un bucle que no está relacionado con colecciones (sino que se dedique a realizar algún otro tipo de acción de manera repetida), el bucle for-each no resulta útil. Habrá que elegir entre el bucle for y el bucle while. El bucle for-each es solo para colecciones.
- El bucle for es adecuado si sabemos al principio del bucle cuántas iteraciones necesitamos (es decir, cuántas veces tenemos que ejecutar el bucle). Esta información puede estar almacenada en una variable, pero no debe cambiar durante la ejecución del bucle. También resulta muy conveniente cuando nos hace falta utilizar de manera explícita el contador del bucle.
- El bucle while es preferible si, al principio del bucle, no sabemos cuántas veces tenemos que ejecutarlo. El final del bucle puede determinarse sobre la marcha mediante alguna condición (por ejemplo, leyendo repetidamente la línea de un archivo hasta que alcancemos el final del archivo).
- Si tenemos que eliminar elementos de la colección mientras la recorremos en bucle, conviene utilizar un bucle for con un `Iterator` si se quiere examinar la colección completa, o un bucle while cuando queramos terminar antes de alcanzar el final de la colección.

Ejercicio 4.71 Corrija todos los errores que haya en el siguiente método.

```
/**
 * Imprimir todos los valores de la matriz marks que
 * sean mayores que mean.
 * @param marks Una matriz de valores que indican marcas.
 * @param mean La marca media (promedio).
 */
public void printGreater(double marks, double mean)
{
    for(index = 0; index <= marks.length; index++) {
        if(marks[index] > mean) {
            System.out.println(marks[index]);
        }
    }
}
```

Ejercicio 4.72 Modifique la clase `LogAnalyzer` para incluir un constructor que pueda admitir el nombre del archivo de registro que hay que analizar. Haga que este constructor pase el nombre del archivo al constructor de la clase `LogfileReader`. Utilice la clase `LogfileCreator` para crear su propio archivo de entradas de registro aleatorias y analizar los datos.

Ejercicio 4.73 Complete el método `numberOfAccesses`, que se muestra a continuación, de modo que cuente el número total de accesos contenidos en el archivo de registro. Complételo utilizando un bucle for para iterar a través de `hourCounts`:

```
/**
 * Devuelve el número de accesos almacenado
 * en el archivo de registro.
 */
public int numberOfAccesses()
{
    int total = 0;
    // Sumar a total el valor de cada elemento
    // de hourCounts.
    ...
    return total;
}
```

Ejercicio 4.74 Añada su método `numberOfAccesses` a la clase `LogAnalyzer` y compruebe que proporciona el resultado correcto. *Sugerencia*: puede simplificar la comprobación haciendo que el analizador lea archivos de registro que contengan solo unas pocas líneas de datos. De esta forma, le resultará más fácil determinar si el método está proporcionando la respuesta correcta. La clase `LogfileReader` tiene un constructor con la siguiente cabecera, para leer de un archivo determinado.

```
/**
 * Crear un LogfileReader que suministre datos
 * a partir de un archivo de registro especificado.
 * @param filename El archivo con los datos de registro.
 */
public LogfileReader(String filename)
```

Ejercicio 4.75 Añada un método `busiestHour` a `LogAnalyzer` que devuelva la hora con la mayor carga de tráfico. Puede hacer esto examinando la matriz `hourCounts` para encontrar el elemento que tenga asociado el mayor valor. *Sugerencia:* ¿es necesario comprobar todos los elementos para ver si hemos encontrado la hora de mayor tráfico? En caso afirmativo, utilice un bucle for o un bucle for-each. ¿Cuál es mejor en este caso?

Ejercicio 4.76 Añada un método `quietestHour` a `LogAnalyzer` que devuelva la hora con la menor carga de tráfico. *Nota:* esto suena casi idéntico al ejercicio anterior, pero hay un pequeña trampa para los despistados. Asegúrese de comprobar su método con algunos datos en los que todas las horas tengan asociado un valor distinto de cero.

Ejercicio 4.77 ¿Qué hora devolverá el método `busiestHour` que ha escrito si hay más de una hora con el mayor valor de número de accesos; es decir, si hay dos horas empatadas en el primer lugar?

Ejercicio 4.78 Añada un método a `LogAnalyzer` que averigüe qué periodo de dos horas es el de mayor tráfico. Devuelva el valor de la primera hora de ese periodo.

Ejercicio 4.79 *Ejercicio avanzado* Guarde el proyecto *weblog-analyzer* con un nombre diferente, para poder desarrollar una nueva versión que realice un análisis más exhaustivo de los datos disponibles. Por ejemplo, sería útil saber qué días tienden a ser más tranquilos que otros, por ejemplo, ¿existe algún tipo de patrón cíclico semanal? Para realizar análisis de datos diarios, mensuales o anuales, tendrá que realizar algunos cambios en la clase `LogEntry`. Esta clase ya se encarga de almacenar todos los valores de una única línea

de registro, pero solo están disponibles la hora y los minutos a través de métodos selectores. Añada métodos adicionales que permitan acceder a los campos restantes de una forma similar. Después, añada al analizador una serie de métodos de análisis adicionales.

Ejercicio 4.80 *Ejercicio avanzado* Si ha completado el ejercicio anterior, puede tratar de ampliar el formato del archivo de registro con campos numéricos adicionales. Por ejemplo, los servidores suelen almacenar un código numérico que indica si un acceso ha tenido éxito o no. El valor 200 representa un acceso con éxito, 403 indica que el acceso al documento estaba prohibido y 404 significa que no se ha podido encontrar el documento. Haga que el analizador proporcione información con el número de accesos que han tenido éxito o han fallado. Este ejercicio puede ser bastante complicado, ya que tendrá que realizar cambios en todas las clases del proyecto.

4.17 Resumen

En este capítulo, hemos hablado de los mecanismos para almacenar colecciones de objetos, en lugar de almacenar objetos independientes en campos separados. Hemos examinado en detalle dos colecciones distintas: `ArrayList` como ejemplo de una colección de tamaño flexible y las matrices como colecciones de tamaño fijo.

La utilización de colecciones como estas será muy importante en todos los proyectos a partir de ahora. Verá que casi todas la aplicaciones necesitan en algún lugar algún tipo de colección. Las colecciones son fundamentales para escribir programas.

Al utilizar colecciones surge la necesidad de iterar a través de los elementos de las mismas, para hacer uso de todos los objetos que contienen. Para este propósito, hemos visto que se usan bucles e iteradores.

Los bucles son también un concepto fundamental en el campo de la computación; son un concepto que tendrá que utilizar en todos los proyectos a partir de ahora. Asegúrese de familiarizarse lo suficiente con la escritura de bucles –no podrá ir muy lejos sin ellos. Al decidir el tipo de bucle que utilizar en una situación concreta, a menudo resultará útil considerar si la tarea implica una iteración definida o indefinida. ¿Hay certidumbre o incertidumbre acerca del número de iteraciones que harán falta?

Además, hemos mencionado la librería de clases de Java, que es una gran colección de clases útiles que podemos emplear para hacer que nuestras propias clases sean más potentes. Tendremos que estudiar la librería con algo más de detalle para ver qué otras cosas contiene que debamos conocer. Este será el tema del siguiente capítulo.

Términos introducidos en el capítulo

colección, matriz, iterador, bucle for-each, bucle while, bucle for, índice, instrucción de importación, librería, paquete, objeto anónimo, iteración definida, iteración indefinida

Resumen de conceptos

- **colección** Un objeto colección puede almacenar un número arbitrario de otros objetos.
- **bucle** Un bucle se puede utilizar para ejecutar un bloque de instrucciones repetidamente, sin tener que escribirlas múltiples veces.
- **iterador** Un iterador es un objeto que proporciona funcionalidad para iterar a través de todos los elementos de una colección.
- **null** La palabra reservada `null` de Java se utiliza para indicar "ningún objeto" cuando una variable de objeto no está haciendo referencia actualmente a ningún objeto concreto. Todo campo que no haya sido explícitamente inicializado contendrá el valor `null` de manera predeterminada.
- **matriz** Una matriz es un tipo especial de colección que puede almacenar un número fijo de elementos.

Ejercicio 4.81 En el proyecto *lab-classes* del que hemos hablado en los capítulos anteriores, la clase `LabClass` incluye un campo `students` para mantener una colección de objetos `Student`. Lea la clase `LabClass` para reforzar algunos de los conceptos que hemos explicado en este capítulo.

Ejercicio 4.82 La clase `LabClass` impone un límite al número de estudiantes que pueden estar matriculados en un grupo tutorial particular. A la vista de esto, ¿cree que sería más apropiado utilizar una matriz de tamaño fijo en lugar de una colección de tamaño flexible pare el campo `students`? Proporcione razones a favor y en contra de ambas alternativas.

Ejercicio 4.83 Escriba de nuevo el método `listAllFiles` de la clase `MusicOrganizer` de *music-organizer-v3* utilizando un bucle for en lugar de un bucle for-each.

Ejercicio 4.84 Java proporciona otro tipo de bucle: el *bucle do-while*. Averigüe cómo funciona este bucle y descríbalo. Escriba un ejemplo de bucle do-while que imprima los números de 1 a 10. Para conseguir información acerca de este bucle, localice una descripción del lenguaje Java, por ejemplo, en la sección "Control Flow Statements" (instrucciones de control de flujo) en

`http://download.oracle.com/javase/tutorial/java/nutsandbolts/`

Ejercicio 4.85 Escriba de nuevo el método `listAllFiles` de la clase `MusicOrganizer` de *music-organizer-v3* utilizando un bucle do-while en lugar de un bucle for-each. Compruebe su solución con cuidado. ¿Funciona correctamente si la colección está vacía?

Ejercicio 4.86 *Ejercicio avanzado* Escriba de nuevo el método `findFirst` de la clase `MusicOrganizer` class en *music-organizer-v4* utilizando un bucle do-while en lugar de un bucle while. Compruebe su solución con cuidado. Pruebe a hacer búsquedas que tengan éxito y otras que fallen. Pruebe a hacer búsquedas en las que el archivo que hay que localizar se encuentre el primero en la lista y otras en las que se encuentre en último lugar.

Ejercicio 4.87 Localice información acerca de la *instrucción switch-case* de Java. ¿Cuál es su propósito? ¿Cómo se utiliza? Escriba un ejemplo. (Esta es también una *instrucción de control de flujo*, por lo que podrá encontrar información acerca de ella en los mismos sitios que para el *bucle do-while*.)

CAPÍTULO

5

Comportamientos más sofisticados

Principales conceptos explicados en el capítulo:

- utilización de clases de librería
- lectura de documentación
- escritura de documentación

Estructuras Java explicadas en este capítulo:

`String`, `ArrayList`, `Random`, `HashMap`, `HashSet`, `Iterator`, `Arrays`, `static`, `final`

En el Capítulo 4, hemos presentado la clase `ArrayList` de la librería de clases de Java. Hemos visto cómo esta clase nos permitía hacer cosas que de otro modo sería difícil de conseguir (en este caso, almacenar un número arbitrario de objetos).

Esa clase era un simple ejemplo de las múltiples clases útiles que contiene la librería Java. La librería está compuesta por miles de clases, muchas de las cuales serán por lo general útiles para nuestro trabajo de programación, aunque también hay disponibles muchas otras que probablemente nunca vayamos a utilizar.

Para un buen programador de Java, es esencial ser capaz de trabajar con la librería Java y tomar decisiones juiciosas acerca de qué clases utilizar. Una vez que haya empezado a trabajar con la librería, comprobará rápidamente que le permite realizar muchas tareas de manera mucho más fácil que si no dispusiera de ella. Aprender a trabajar con las clases de librería es el tema principal de este capítulo.

Los elementos de la librería no son solo un conjunto de clases arbitrarias y no relacionadas que tengamos que aprender individualmente, sino que a menudo están organizadas formando relaciones y aprovechando características comunes. Aquí, volvemos a encontrarnos con el concepto de abstracción, que nos ayudará a tratar con grandes cantidades de información. Una de las partes más importante de la librería son las colecciones, de las que la clase `ArrayList` es solo un ejemplo. Veremos otros tipos de colecciones en este capítulo y comprobaremos que comparten muchos atributos, por lo que a menudo podremos abstraernos de los detalles específicos de una colección concreta y hablar de clases de colección en general.

Presentaremos y explicaremos nuevas clases de colección, así como algunas otras clases de utilidad. A lo largo del capítulo, trabajaremos en la construcción de una única aplicación (el

sistema *TechSupport*), que hace uso de distintas clases de librería. En los proyectos del libro se incluye una implementación completa de todas las ideas y del código fuente analizados aquí, así como varias versiones intermedias. Aunque esto le permitirá estudiar la solución completa, le animamos a que realice los ejercicios incluidos en el capítulo. Estos ejercicios comenzarán, después de una breve ojeada al programa completo, con una versión inicial muy simple del proyecto, para ir luego desarrollando e implementando la solución completa.

La aplicación hace uso de varias nuevas técnicas y clases de librería –cada una de las cuales requiere un estudio individual–, como mapas hash, conjuntos, extracción de símbolos en cadenas de caracteres y usos avanzados de los números aleatorios. Debe tener en cuenta que este no es un capítulo para ser leído y comprendido en un solo día, sino que contiene varias secciones, cada una de las cuales merece varios días seguidos de estudio. En conjunto, cuando llegue al final del capítulo y haya conseguido llevar a cabo la implementación sugerida en los ejercicios, habrá adquirido una gran cantidad de conocimientos acerca de varios temas importantes.

5.1 Documentación para clases de librería

La librería Java es enorme. Esta formada por miles de clases, cada una de las cuales tiene muchos métodos, que a su vez pueden tener o no parámetros y pueden tener o no tipos de retorno. Es imposible memorizar todos los métodos y todos los detalles correspondientes a cada uno. En lugar de ello, lo que un buen programador Java debe hacer es:

> **Concepto:**
>
> **Librería Java**
> La **librería de clases estándar** de Java contiene muchas clases de gran utilidad. Es importante saber cómo utilizar la librería.

- Conocer por su nombre algunas de las clases más importantes y sus métodos (`ArrayList` es una de esas clases importantes).
- Saber localizar información acerca de esas clases y buscar los correspondientes detalles (como por ejemplo métodos y parámetros).

En este capítulo, presentaremos algunas de las clases más importantes de la librería de clases, y en el resto del libro hablaremos de otras clases adicionales de la librería. Pero lo más importante es que le mostraremos cómo puede explorar y comprender la librería por sí mismo. Esto le permitirá escribir programas mucho más interesantes. Afortunadamente, la librería Java está muy bien documentada. Esta documentación está disponible en formato HTML (por lo que se puede leer en un explorador web), que será la aplicación que utilizaremos para conseguir información acerca de las clases de librería.

Leer y comprender la documentación es la primera etapa de nuestra introducción a las clases de librería. Llevaremos este enfoque un poco más allá y veremos también cómo preparar nuestras propias clases para que otras personas también puedan utilizarlas de la misma forma que usarían las clases de la librería estándar. Esto tiene una gran importancia para el desarrollo de software en el mundo real, donde los equipos de trabajo tienen que tratar con proyectos de gran envergadura y tienen que encargarse del mantenimiento del software a lo largo del tiempo.

Una cosa que puede haber observado acerca de la clase `ArrayList` es que la hemos utilizado sin haber echado siquiera un vistazo a su código fuente. No nos molestamos en comprobar cómo estaba implementada, porque no lo necesitábamos para poder hacer uso de su funcionalidad. Lo único que necesitábamos saber era el nombre de la clase, los nombres de los métodos, los parámetros y tipos de retorno de esos métodos y qué es lo que esos métodos

hacen exactamente. Realmente no nos preocupaba cómo se llevaba a cabo ese trabajo. Esta situación es típica a la hora de utilizar clases de librería.

Lo mismo cabe decir de esas clases de proyectos software de gran tamaño. Normalmente, varias personas trabajan juntas en un proyecto encargándose de distintas partes del mismo. Cada programador debe concentrarse en su propia área y no necesita comprender los detalles de las restantes partes (hemos hablado de esto en la Sección 3.2, al presentar los conceptos de abstracción y modularización). De hecho, cada programador debe ser capaz de utilizar las clases de otros miembros del equipo como si fueran clases de librería, haciendo un uso juicioso de ellas, pero sin necesidad de saber cómo funcionan internamente.

Para que esto funcione, cada miembro del equipo debe escribir documentación acerca de su clase de forma similar a la documentación que existe para la librería estándar de Java; esta documentación permite a otras personas utilizar una clase sin necesidad de leer su código. También hablaremos de este tema a lo largo del capítulo.

5.2 El sistema *TechSupport*

Como siempre, exploraremos estas cuestiones con un ejemplo. Esta vez, utilizaremos la aplicación *TechSupport*. Puede encontrarla en los proyectos del libro con el nombre de *tech-support1*.

TechSupport es un programa que pretende proporcionar soporte técnico para los clientes de una compañía de software ficticia denominada DodgySoft. Hace tiempo, DodgySoft tenía un departamento de soporte técnico, con personas atendiendo una serie de teléfonos. Los clientes podían llamar para pedir consejo y solicitar ayuda relativa a sus problemas técnicos con los productos software de DodgySoft. Sin embargo, recientemente, los negocios no han estado yendo tan bien, por lo que DodgySoft decidió deshacerse de su departamento de soporte técnico con el fin de ahorrar dinero. Ahora, quieren desarrollar el sistema *TechSupport* para dar la impresión de que se sigue proporcionando soporte. Se supone que el sistema debe simular las respuestas que una persona de soporte técnico daría. Los clientes pueden comunicarse con el sistema de soporte técnico a través de Internet.

5.2.1 Exploración del sistema *TechSupport*

Ejercicio 5.1 Abra y ejecute el proyecto *tech-support-complete*. Para ejecutarlo hay que crear un objeto de la clase `SupportSystem` e invocar su método `start`. Introduzca algunos problemas que pueda estar experimentando con su software, con el fin de probar el sistema. Tendrá que describir los problemas en inglés. Compruebe cómo se comporta el sistema. Escriba "bye" cuando termine. No es necesario que examine el código fuente por el momento. Este proyecto es la solución completa que habremos desarrollado al finalizar el capítulo. El propósito de este ejercicio es solo que se haga una idea de qué es lo que queremos conseguir.

Eliza La idea del proyecto TechSupport está basada en el pionero programa de inteligencia artificial Eliza, desarrollado por Joseph Weizenbaum en el Instituto Tecnológico de Massachusetts en la década de 1960. Puede conseguir más información acerca del programa original buscando en la web por "Eliza" y "Weizenbaum."

Figura 5.1
Un primer diálogo de `TechSupport`.

```
BlueJ: Terminal Window - tech-support1
Welcome to the DodgySoft Technical Support System.

Please tell us about your problem.
We will assist you with any problem you might have.
Please type 'bye' to exit our system.
> My system crashes all the time.
That sounds interesting. Tell me more...
> I am using Windows 9. Should your program run on Windows 9?
That sounds interesting. Tell me more...
> I really need help! It's not working.
That sounds interesting. Tell me more...
> Why are you always saying "That sounds interesting"?
That sounds interesting. Tell me more...
> I hate you.
That sounds interesting. Tell me more...
> bye
Nice talking to you. Bye...
```

Ahora comenzaremos nuestra exploración más detallada utilizando el proyecto *tech-support1*. Se trata de una primera implementación rudimentaria de nuestro sistema, implementación que iremos mejorando a lo largo del capítulo. De esta forma, podremos conseguir una mejor comprensión del sistema en su conjunto que si nos limitáramos a leer la solución completa.

En el Ejercicio 5.1 hemos visto que el programa básicamente se dedica a mantener un diálogo con el usuario. El usuario puede escribir una pregunta y el sistema responde. Pruebe a hacer lo mismo con nuestra versión del proyecto *tech-support1*.

En la versión completa de *TechSupport*, el sistema se las arregla para generar respuestas razonablemente variadas –¡en ocasiones, incluso parecen tener sentido! En la versión prototipo que estamos utilizando como punto de partida, las respuestas son mucho más restringidas (Figura 5.1). Comprobará rápidamente que la respuesta es siempre la misma:

"That sounds interesting. Tell me more..." ("Parece interesante. Cuénteme más...")

Esto, de hecho, no resulta interesante en absoluto, y tampoco es muy convincente a la hora de tratar de simular que tenemos una persona de soporte técnico al otro lado de la pantalla. En breve, trataremos de mejorar este aspecto. Sin embargo, antes de hacerlo, vamos a explorar con más detalle lo que hasta ahora tenemos.

El diagrama del proyecto nos muestra tres clases: `SupportSystem`, `InputReader` y `Responder` (Figura 5.2). `SupportSystem` es la clase principal, que utiliza `InputReader` para obtener entradas desde el terminal y utiliza `Responder` para generar una respuesta.

Examine con más detalle la clase `InputReader` creando un objeto de esta clase y luego echando un vistazo a los métodos de ese objeto. Verá que solo tiene un método disponible,

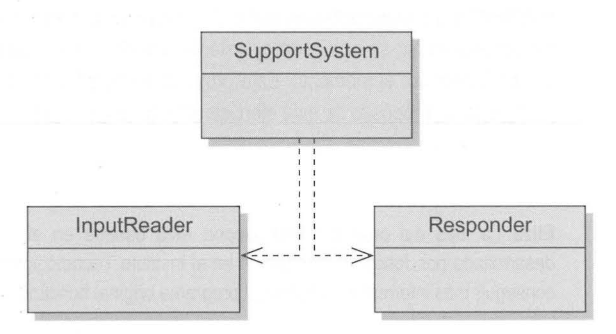


Figura 5.2
Diagrama de clases de `TechSupport`.

denominado `getInput`, que devuelve una cadena de caracteres. Pruebe ese método. El método permite escribir una línea de entrada en el terminal y luego devuelve como resultado del método lo que fuera que hayamos escrito. No vamos a analizar por el momento cómo funciona esto internamente, así que quédese con la idea de que `InputReader` tiene un método `getInput` que devuelve una cadena de caracteres.

Haga lo mismo con la clase `Responder`. Comprobará que tiene un método `generateResponse` que siempre devuelve la cadena `"That sounds interesting. Tell me more..."`. Esto explica lo que hemos visto anteriormente en el diálogo.

Ahora examinemos con más detalle la clase `SupportSystem`.

5.2.2 Lectura del código

El código fuente completo de la clase `SupportSystem` se muestra en el Código 5.1. El Código 5.2 muestra el código fuente de la clase `Responder`.

Código 5.1
El código fuente de `SupportSystem`.

```
/**
 * Esta clase implementa un sistema de soporte técnico.
 * Es la clase principal de este proyecto.
 * El sistema de soporte se comunica mediante entrada/salida
 * de texto a través del terminal de texto.
 * Esta clase utiliza un objeto de la clase InputReader para
 * leer la entrada del usuario y un objeto de la clase Responder
 * para generar respuestas.
 * Contiene un bucle que lee repetidamente la entrada y
 * genera la correspondiente salida hasta que el usuario
 * indica que quiere salir.
 *
 * @author     Michael Kölling y David J. Barnes
 * @version    0.1 (2011.07.31)
 */

public class SupportSystem
{
    private InputReader reader;
    private Responder responder;

    /**
     * Crea un sistema de soporte técnico.
     */
    public SupportSystem()
    {
        reader = new InputReader();
        responder = new Responder();
    }

    /**
     * Inicia el sistema de soporte técnico.
```

Código 5.1
(continuación)
El código fuente de `SupportSystem`.

```
     * Imprimirá un mensaje de bienvenida y entablará un
     * diálogo con el usuario, hasta que el usuario dé por
     * terminado el diálogo.
     */
    public void start()
    {
        boolean finished = false;

        printWelcome();
        while(!finished) {
            String input = reader.getInput();

            if(input.startsWith("bye")) {
                finished = true;
            }
            else {
                String response = responder.generateResponse();
                System.out.println(response);
            }
        }
        printGoodbye();
    }

    /**
     * Presenta un mensaje de bienvenida en la pantalla.
     */
    private void printWelcome()
    {
        System.out.println(
            "Welcome to the DodgySoft Technical Support System.");
        System.out.println();
        System.out.println("Please tell us about your problem.");
        System.out.println(
            "We will assist you with any problem you might have.");
        System.out.println(
            "Please type 'bye' to exit our system.");
    }

    /**
     * Presenta un mensaje de despedida en la pantalla.
     */
    private void printGoodbye()
    {
        System.out.println("Nice talking to you. Bye...");
    }
}
```

Código 5.2
El código fuente de `Responder`.

```
/**
 * Esta clase representa un objeto generador de respuestas.
 * Se utiliza para generar una respuesta automática a cada
 * cadena de entrada.
 *
 * @author     Michael Kölling y David J. Barnes
 * @version    0.1 (2011.07.31)
 */

public class Responder
{
    /**
     * Construir un Responder - no hay nada que hacer
     */
    public Responder()
    {
    }

    /**
     * Generar una respuesta.
     * @return   Una cadena de caracteres que hay que mostrar
     *           como respuesta
     */
    public String generateResponse()
    {
        return "That sounds interesting. Tell me more...";
    }
}
```

Examinando el Código 5.2, vemos que la clase `Responder` es trivial. Tiene solo un método, que siempre devuelve la misma cadena de caracteres. Esto es algo que mejoraremos más adelante. Por el momento, vamos a concentrarnos en la clase `SupportSystem`.

`SupportSystem` declara dos campos de instancia para almacenar un objeto `InputReader` y otro objeto `Responder`, y asigna esos dos objetos dentro de su constructor.

Al final, tiene dos métodos denominados `printWelcome` y `printGoodbye`. Estos métodos simplemente imprimen un texto –un mensaje de bienvenida y un mensaje de despedida, respectivamente.

La parte más interesante del código es el método que se encuentra hacia la mitad del listado: `start`. Veremos este método más detalladamente.

Al principio del método hay una llamada a `printWelcome` y al final hay otra llamada a `printGoodbye`. Estas dos llamadas se encargan de imprimir los correspondientes mensajes de texto en los momentos apropiados. El resto del método consta de una declaración de una variable booleana y de un bucle while. La estructura es:

```
boolean finished = false;
while(!finished) {
    hacer algo
    if(condición de salida) {
        finished = true;
    }
    else {
        hacer algo más
    }
}
```

Este patrón de código es una variante de la estructura de bucle while que hemos explicado en la Sección 4.10. Utilizamos `finished` como indicador que toma el valor `true` cuando queremos finalizar el bucle (y con él, el programa completo). Por supuesto, nos aseguramos de que ese indicador tome inicialmente el valor `false`. (Recuerde que el signo de exclamación es un operador *not*.)

La parte principal del bucle (la parte que se repite una y otra vez mientras deseamos continuar) está compuesta por tres instrucciones, si dejamos de lado la comprobación de la condición de salida:

```
String input = reader.getInput();
...
String response = responder.generateResponse();
System.out.println(response);
```

Por tanto, lo que el bucle hace repetidamente es:

- leer una entrada de usuario,
- pedir al generador de respuestas que genere una respuesta nueva e
- imprimir esa respuesta.

(Ya habrá observado que la respuesta no depende de la entrada en absoluto. Por supuesto, algo que tendremos que mejorar más adelante.)

La última parte que tenemos que examinar es la comprobación de la condición de salida. La intención es que el programa termine cuando el usuario escriba la palabra *bye*. La sección relevante del código fuente de la clase es la siguiente:

```
String input = reader.getInput();
if(input.startsWith("bye")) {
    finished = true;
}
```

Si es capaz de entender estos fragmentos aisladamente, entonces conviene que examine de nuevo el método `start` completo en el Código 5.1 para ver si puede comprender cómo funciona todo el conjunto.

En el último fragmento de código que acabamos de examinar, se utiliza un método denominado `startsWith`. Puesto que ese método se invoca sobre la variable `input`, que almacena un objeto `String`, debe ser un método de la clase `String`. ¿Pero qué es lo que hace ese método? ¿Y cómo podemos averiguarlo?

Podemos adivinar, simplemente viendo el nombre del método, que el método comprueba si la cadena de entrada comienza con la palabra "bye". Podemos verificar esta suposición haciendo algunos experimentos. Ejecute de nuevo el sistema *TechSupport* y escriba "bye bye" o "bye everyone". Observará que ambas versiones hacen que el programa termine. Sin embargo, observe que si escribimos "Bye" o " bye" –comenzando con una letra mayúscula o con espacio delante de la palabra– entonces no se reconoce que la cadena comience por "bye". Esto podría resultar algo molesto para el usuario, pero en realidad podemos resolver estos problemas con solo saber algo más acerca de la clase `String`.

¿Cómo podemos encontrar más información acerca del método `startsWith` o de los restantes métodos de la clase `String`?

5.3 Lectura de la documentación de las clases

La clase `String` es una de las clases de la librería estándar de clases Java. Podemos conocer más detalles acerca de la misma leyendo la documentación de librería para la clase `String`.

Concepto:

La **documentación de la librería de clases** Java muestra detalles acerca de todas las clases de la librería. La utilización de esta documentación es esencial para poder hacer un buen uso de las clases de librería.

Para ello, seleccione el elemento *Java Class Libraries* del menú *Help* de BlueJ. Se abrirá un explorador web mostrando la página principal de la documentación de la API (*Application Programming Interface*, Interfaz de programación de aplicaciones) de Java.[1]

El explorador web mostrará tres marcos. En el marco situado en la parte superior izquierda, podrá ver una lista de paquetes. Debajo de él verá una lista de todas las clases de la librería Java. El marco más grande situado a la derecha se emplea para mostrar los detalles de un paquete o clase seleccionado.

En la lista de clases de la izquierda, localice y seleccione la clase `String`. El marco de la derecha mostrará entonces la documentación de la clase `String` (Figura 5.3).

Ejercicio 5.2 Investigue la documentación de `String`. Después mire la documentación correspondiente a otras clases. ¿Cuál es la estructura de la documentación de las clases? ¿Qué secciones son comunes a todas las descripciones de clases? ¿Cuál es su propósito?

Ejercicio 5.3 Busque el método `startsWith` en la documentación de `String`. Existen dos versiones. Describa con sus palabras qué es lo que hacen y las diferencias que existen entre ellas.

Ejercicio 5.4 ¿Hay algún método en la clase `String` que compruebe si una cadena termina con un sufijo determinado? En caso afirmativo, ¿cómo se llama y cuáles son sus parámetros y su tipo de retorno?

Ejercicio 5.5 ¿Existe un método en la clase `String` que devuelva el número de caracteres de una cadena? En caso afirmativo, ¿cómo se llama y cuáles son sus parámetros?

---

[1] De manera predeterminada, esta función accede a la documentación a través de Internet. Esto no funcionará si su máquina no tiene acceso a red. Sin embargo, BlueJ puede configurarse para que utilice una copia local de la documentación de la API Java. Le recomendamos que haga esto, porque acelera el acceso a la documentación y puede funcionar sin conexión a Internet. Para conocer más detalles, consulte el Apéndice A.

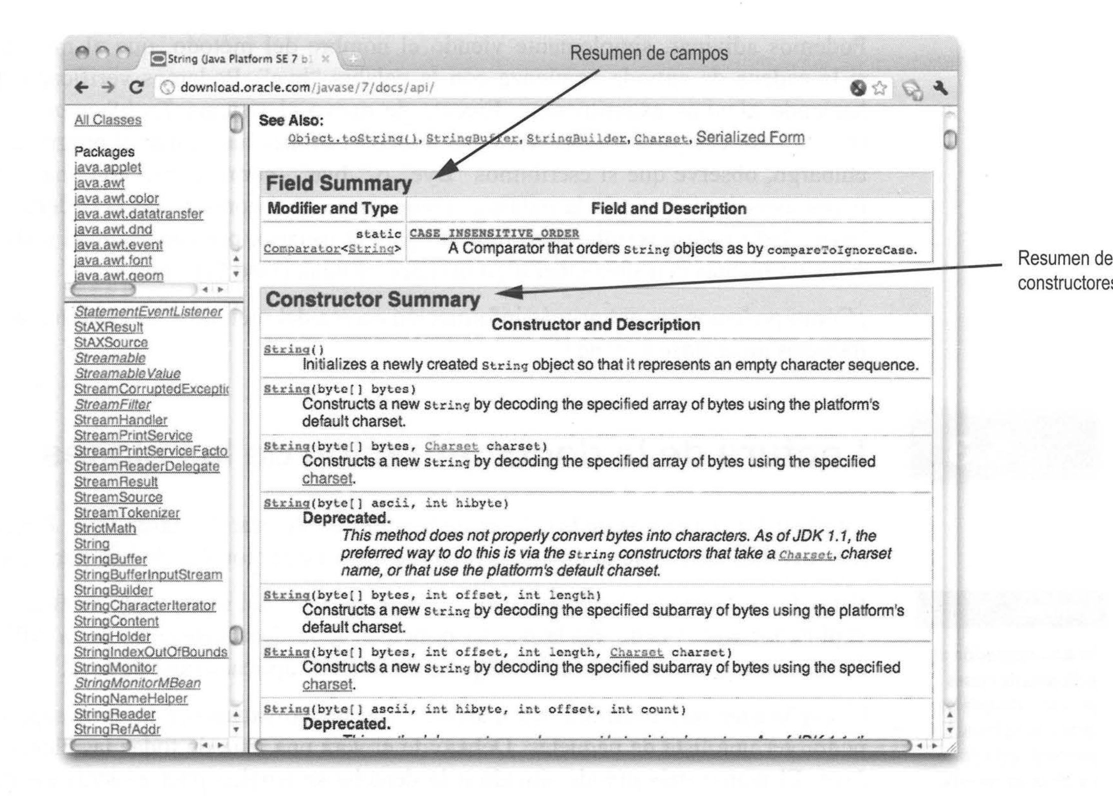


Figura 5.3 La documentación de la librería de clases Java.

Ejercicio 5.6 Si ha encontrado métodos para las dos tareas indicadas anteriormente, ¿cómo los ha encontrado? ¿Es fácil o difícil encontrar los métodos que se buscan? ¿Por qué?

5.3.1 Interfaces e implementación

Verá que la documentación incluye diferentes elementos de información. Entre otros, se incluyen los siguientes:

- El nombre de la clase.
- Una descripción general del propósito de la clase.
- Una lista de los constructores y métodos de la clase.
- Los parámetros y los tipos de retorno para cada constructor y método.
- Una descripción del propósito de cada constructor y método.

Concepto:

La **interfaz** de una clase describe lo que una clase hace y cómo se la puede utilizar sin mostrar su implementación.

Esta información, tomada conjuntamente, se denomina *interfaz* de una clase. Observe que la interfaz *no* muestra el código fuente que implementa a la clase. Si una clase está bien descrita (es decir, si su interfaz está bien escrita), entonces el programador no necesita ver el código fuente para ser capaz de utilizar la clase. Con ver la interfaz, tenemos toda la información necesaria. Esto es de nuevo un ejemplo del concepto de abstracción.

Concepto:

El código fuente completo que define una clase es la **implementación** de dicha clase.

El código fuente subyacente, que es el que hace que la clase funcione, se conoce como *implementación* de la clase. Normalmente, un programador trabaja en la implementación de una clase a la vez que hace uso de otras diversas clases a través de sus interfaces.

Esta distinción entre la interfaz y la implementación es un concepto muy importante, que volverá a aparecer una y otra en este capítulo y en capítulos posteriores.

Nota La palabra *interfaz* tiene varios significados en el campo de la programación y en el contexto del lenguaje Java. Se utiliza para describir la parte públicamente visible de una clase (que es como hemos estado utilizando el concepto aquí), pero también tiene otros significados. La interfaz de usuario (a menudo una interfaz gráfica de usuario) se denomina en ocasiones simplemente *interfaz*, pero Java también tiene una estructura de lenguaje denominada *interface* (de la que hablaremos en el Capítulo 10) que está relacionada con el concepto de interfaz, pero que es diferente del significado que le hemos dado aquí.

Es importante distinguir entre los distintos significados de la palabra *interfaz* en cada contexto concreto.

El término interfaz también se utiliza para métodos individuales. Por ejemplo, la documentación de `String` nos muestra la interfaz del método `length`:

```
public int length()
```

Returns the length of this string. The length is equal to the number of Unicode code units in the string.

Specified by:

length in interface CharSequence

Returns:

the length of the sequence of characters represented by this object.

La interfaz de un método consta de la *signatura* del método y de un comentario (mostrado aquí en cursiva). La signatura de un método incluye (en este orden):

- Un modificador de acceso (que aquí es `public`), del que hablaremos más adelante.
- El tipo de retorno del método (en este caso `int`).
- El nombre del método.
- Una lista de parámetros (que en este ejemplo está vacía).

La interfaz proporciona todo lo que necesitamos conocer para hacer uso de este método.

5.3.2 Utilización de métodos de las clases de librería

Volvamos a nuestro sistema *TechSupport*. Ahora queremos mejorar un poco el procesamiento de la entrada. Hemos visto en las explicaciones anteriores que nuestro sistema no es muy tolerante: si escribimos "Bye" o " bye" en lugar de "bye", por ejemplo, la palabra no se reconoce. Queremos cambiar esto ajustando el texto leído de un usuario de modo que todas esas variantes sean reconocidas como "bye".

La documentación de la clase `String` nos dice que dispone de un método denominado `trim` para eliminar espacios al principio y al final de la cadena de caracteres. Podemos utilizar dicho método para resolver el segundo de los casos problemáticos.

Concepto:

Objetos inmutables Se dice que un objeto es inmutable si su contenido o estado no puede cambiarse después de crearlo. Las cadenas de caracteres son un ejemplo de objeto inmutable.

Ejercicio 5.7 Localice el método `trim` en la documentación de la clase `String`. Escriba la signatura de dicho método. Escriba un ejemplo de llamada a dicho método con una variable de tipo `String` denominada `text`.

Un detalle importante acerca de los objetos String es que son *inmutables*; es decir, no pueden modificarse después de haberlos creado. Fíjese especialmente en que el método `trim`, por ejemplo, devuelve una nueva cadena de caracteres, no modifica la cadena original. Preste especial atención al siguiente comentario de "Error común".

Error común Es un error común en Java tratar de modificar una cadena. Por ejemplo, escribiendo

```
input.toUpperCase();
```

Esto es incorrecto (las cadenas de caracteres no pueden modificarse), aunque lamentablemente esto no produce ningún error. La instrucción simplemente no tienen ningún efecto, y la cadena de entrada no será modificada.

El método `toUpperCase`, así como otros métodos de cadena no modifica la cadena original, sino que devuelve una *nueva* cadena que es similar a la original, pero con algunos cambios aplicados (en este caso, los caracteres se han pasado a mayúscula). Si queremos modificar nuestra variable de entrada, entonces tenemos que asignar otra vez este nuevo objeto a la variable (descartando la original), como en el siguiente ejemplo:

```
input = input.toUpperCase();
```

El nuevo objeto también podría asignarse a otra variable o procesarse de alguna otra manera.

Después de estudiar la interfaz del método `trim`, podemos ver que se pueden eliminar los espacios de una cadena de entrada con la siguiente línea de código:

```
input = input.trim();
```

Este código solicitará al objeto `String` almacenado en la variable `input` que cree una nueva cadena, similar a la anterior, pero sin los espacios iniciales y finales. El nuevo objeto `String` se almacena entonces en la variable `input`, porque no tenemos ningún uso adicional que dar a la cadena de caracteres anterior. Por tanto, después de esta línea de código, `input` hace referencia a una cadena que no tiene espacios ni al principio ni al final.

Ahora podemos insertar esta línea en nuestro código fuente, con lo que quedará:

```
String input = reader.getInput();
input = input.trim();

if(input.startsWith("bye")) {
    finished = true;
}
else {

      ...  Código omitido.

}
```

Las dos primeras líneas pueden agruparse en un única línea:

```
String input = reader.getInput().trim();
```

El efecto de esta línea de código es idéntico al de las dos primeras líneas anteriores. El lado derecho debe leerse como si tuviera paréntesis, de la forma siguiente:

```
(reader.getInput()) . trim()
```

Qué versión prefiera cada uno es, básicamente, una cuestión de gusto personal. La decisión debe tomarse principalmente pensando en la legibilidad: utilice la versión que le resulte más fácil de leer y de entender. A menudo, los programadores inexpertos preferirán la versión en dos líneas, mientras que los más experimentados están acostumbrados al estilo en una línea.

Ejercicio 5.8 Implemente esta mejora en su versión del proyecto *tech-support1*. Compruébela para confirmar que se pueden meter espacios adicionales sin problemas alrededor de la palabra "bye".

Ahora que hemos resuelto el problema provocado por los espacios iniciales o finales de la entrada, nos queda por solucionar el problema relativo a las letras mayúsculas. Sin embargo, una investigación adicional de la documentación de la clase `String` sugiere una posible solución, ya que en esa documentación se describe un método denominado `toLowerCase`.

Ejercicio 5.9 Mejore el código de la clase `SupportSystem` en el proyecto *tech-support1* de modo que se ignore si la entrada está escrita en mayúsculas o minúsculas. Para hacer esto, utilice el método `toLowerCase` de la clase `String` (que pasa todos los caracteres a minúsculas). Recuerde que este método no modifica en realidad el objeto `String` sobre el que se invoca, sino que provoca la creación de un nuevo objeto con un contenido ligeramente distinto.

5.3.3 Comprobación de la igualdad entre cadenas

Una solución alternativa habría sido comprobar si la cadena de entrada *es* la cadena "bye" en lugar de ver si *comienza con la cadena* "bye". Un intento (¡incorrecto!) de escribir este código tendría el aspecto siguiente:

```
if(input == "bye") { // ¡no funciona siempre!
    ...
}
```

El problema aquí es que es posible que existan varios objetos `String` independientes que representen el mismo texto. Por ejemplo, dos objetos `String` podrían contener los caracteres "bye". El operador de igualdad (==) comprueba si cada lado del operador hace referencia al *mismo objeto*, no si tienen el mismo valor. Son dos cosas completamente distintas.

En nuestro ejemplo, lo que nos interesa es saber si la variable de entrada y la constante de cadena 'bye' representan el mismo valor, no si hacen referencia al mismo objeto. Por tanto, utilizar el operador == es erróneo. Esa comprobación podría devolver el valor `false`, aun cuando el valor de la variable `input` fuera 'bye'.[2]

[2] Lamentablemente, la implementación de las cadenas de Java implica que la utilización de == proporcionará a menudo, de manera confusa, la respuesta "correcta" a la hora de comparar dos objetos `String` diferentes con idéntico contenido. Sin embargo, no debe *nunca* utilizarse == entre objetos `String` cuando lo que se quiera es comparar su contenido.

La solución es utilizar el método `equals`, definido en la clase `String`. Este método comprueba correctamente si el contenido de dos objetos `String` coincide. El código correcto sería:

```
if(input.equals("bye")) {
    ...
}
```

Por supuesto, esto puede combinarse también con los métodos `trim` y `toLowerCase`.

Error común Comparar cadenas con el operador == puede dar lugar a resultados distintos de los pretendidos. Como regla general, las cadenas deberían casi siempre compararse utilizando `equals`, en lugar del operador ==.

Ejercicio 5.10 Localice el método `equals` en la documentación de la clase `String`. ¿Cuál es el tipo de retorno de este método?

Ejercicio 5.11 Modifique su implementación para usar el método `equals` en lugar de `startsWith`.

5.4 Adición de comportamiento aleatorio

Hasta ahora, hemos hecho una pequeña mejora en el proyecto *TechSupport*, pero en conjunto la funcionalidad continúa siendo muy básica. Uno de los problemas principales es que siempre proporciona la misma respuesta, independientemente de la entrada del usuario. Ahora vamos a mejorar esto definiendo un conjunto de frases plausibles con las que responder. Después, haremos que el programa seleccione aleatoriamente una de ellas, cada vez que necesite generar una respuesta. Esto será una extensión de la clase `Responder` de nuestro proyecto.

Para hacer esto, emplearemos un `ArrayList` para almacenar algunas cadenas de respuesta, generar un número entero aleatorio y utilizaremos el número aleatorio como índice para seleccionar en la lista de respuestas una de esas frases. En esta versión, la respuesta seguirá sin depender de la entrada del usuario (posteriormente haremos eso), pero al menos la respuesta variará y el aspecto del programa será mucho mejor.

En primer lugar, tenemos que ver cómo generar un número entero aleatorio.

Aleatorio y pseudoaleatorio La generación de números aleatorios en una computadora no es tan fácil de realizar, de hecho, como inicialmente podría pensarse. Dado que las computadoras operan de un forma bien definida y determinista, que descansa en el hecho de que todos los cálculos son predecibles y repetibles, proporcionan poco espacio para un comportamiento realmente aleatorio.

Los investigadores han propuesto, a lo largo del tiempo, muchos algoritmos para generar secuencias de números aparentemente aleatorias. Estos números normalmente no son realmente aleatorios, sino que se generan siguiendo una serie muy complicada de reglas. Por ello se los denomina números *pseudoaleatorios*.

En un lenguaje como Java, la generación de números pseudoaleatorios está implementada, afortunadamente, en una clase de librería, por lo que lo único que tenemos que hacer para obtener un número pseudoaleatorio es hacer algunas llamadas a la librería.

Si desea leer algo más acerca de este tema, haga una búsqueda web con las palabras "números pseudoaleatorios".

5.4.1 La clase Random

La librería de clases Java contiene una clase denominada `Random` que nos será de ayuda para nuestro proyecto.

Ejercicio 5.12 Localice la clase `Random` en la documentación de la librería de clases Java. ¿En qué paquete se encuentra? ¿Qué es lo que hace? ¿Cómo se construye una instancia? ¿Cómo se genera un número aleatorio? Observe que probablemente no podrá comprender toda la información que se proporciona en la documentación. Trate simplemente de averiguar las cosas que necesita saber.

Ejercicio 5.13 Escriba un pequeño fragmento de código (en papel) que genere un número entero aleatorio utilizando esta clase.

Para generar un número aleatorio, tenemos que:

- crear una instancia de la clase `Random` y
- hacer una llamada a un método de dicha instancia para obtener un número.

Examinando la documentación, vemos que hay varios métodos denominados `next`*Algo* para generar valores aleatorios de distintos tipos. El que genera un número aleatorio entero se denomina `nextInt`.

El siguiente fragmento ilustra el código necesario para generar e imprimir un número aleatorio entero:

```
Random randomGenerator;

randomGenerator = new Random();
int index = randomGenerator.nextInt();
System.out.println(index);
```

Este fragmento de código crea una nueva instancia de la clase `Random` y la almacena en la variable `randomGenerator`. A continuación, llama al método `nextInt` para recibir un número aleatorio, lo almacena en la variable `index` y al final lo imprime.

Ejercicio 5.14 Escriba un código (en BlueJ) para comprobar la generación de números aleatorios. Para hacer esto, cree una nueva clase denominada `RandomTester`. Puede crear esta clase en el proyecto *tech-support1* o puede crear un nuevo proyecto para la misma –no tiene importancia. En la clase `RandomTester`, implemente dos métodos: `printOneRandom` (que imprime un número aleatorio) y `printMultiRandom(int howMany)` (que dispone de un parámetro para especificar cuántos números queremos y luego imprime la cantidad apropiada de números aleatorios).

Su clase solo debe crear un única instancia de la clase `Random` (en su constructor) y almacenarla en un campo. No cree una nueva instancia de `Random` cada vez que desee generar un nuevo número.

5.4.2 Números aleatorios con rango limitado

Los números aleatorios que hemos visto hasta ahora se generaban a partir del rango completo de enteros Java (–2147483648 a 2147483647). Eso está bien para un experimento,

pero rara vez resulta útil. Más frecuentemente, lo que querremos es obtener números aleatorios dentro de un rango limitado específico.

La clase `Random` también ofrece un método para satisfacer esta necesidad. Se llama también `nextInt`, pero tiene un parámetro para especificar el rango de números que nos gustaría usar.

Ejercicio 5.15 Localice el método `nextInt` de la clase `Random` que permite especificar el rango objetivo de números aleatorios. ¿Cuáles son los posibles números aleatorios que se generan cuando invocamos este método con 100 como parámetro?

Ejercicio 5.16 Escriba un método en su clase `RandomTester` denominado `throwDice` que devuelva un número aleatorio comprendido entre 1 y 6 (ambos inclusive), para simular el lanzamiento de un dado.

Ejercicio 5.17 Escriba un método denominado `getResponse` que devuelva aleatoriamente una de las cadenas `"yes"`, `"no"` o `"maybe"`.

Ejercicio 5.18 Amplíe su método `getResponse` para que utilice un `ArrayList` para almacenar un número arbitrario de respuestas y para devolver aleatoriamente una de ellas.

Al utilizar un método que genere números aleatorios a partir de un rango especificado, hay que tener cuidado de comprobar si los límites son *inclusivos* o *exclusivos*. El método `nextInt (int n)` en la clase `Random` de la librería Java, por ejemplo, especifica que genera un número comprendido entre `0` (inclusive) y `n` (exclusive). Esto significa que el valor `0` está incluido en los posibles resultados, mientras que el valor especificado para `n` no lo está. El número más alto que puede devolver una de esas llamadas es `n-1`.

Ejercicio 5.19 Añada un método a su clase `RandomTester` que admita un parámetro `max` y genere un número aleatorio comprendido en el rango entre 1 y `max` (inclusive).

Ejercicio 5.20 Añada un método a su clase `RandomTester` que admita dos parámetros, `min` y `max`, y genere número aleatorio comprendido en el rango entre `min` y `max` (inclusive). Vuelva a escribir el cuerpo del método que haya desarrollado en el ejercicio anterior de manera que ahora llame a este método para generar su resultado. Observe que no es necesario utilizar un bucle en este método.

5.4.3 Generación de respuestas aleatorias

Ahora podemos tratar de ampliar la clase `Responder` para seleccionar una respuesta aleatoria de entre una lista de frases predefinidas. El Código 5.2 muestra el código de la clase `Responder`, en la forma que tiene en nuestra primera versión.

Añadiremos código a esta primera versión para:

- declarar un campo de tipo `Random` para almacenar el generador de números aleatorios;
- declarar un campo de tipo `ArrayList` para almacenar nuestras posibles respuestas;
- crear los objetos `Random` y `ArrayList` en el constructor de `Responder`;
- rellenar la lista de respuestas con algunas frases;
- seleccionar y devolver una frase aleatoria cuando se invoque `generateResponse`.

El Código 5.3 muestra una versión del código fuente de `Responder` con estas adiciones.

Código 5.3
El código fuente de `Responder` con respuestas aleatorias.

```
import java.util.ArrayList;
import java.util.Random;

/**
 * Esta clase representa un objeto generador de respuestas.
 * Se usa para generar una respuesta automática seleccionando
 * aleatoriamente una frase a partir de una lista de
 * respuestas predefinidas.
 *
 * @author    Michael Kölling y David J. Barnes
 * @version   0.2 (2011.07.31)
 */
public class Responder
{
    private Random randomGenerator;
    private ArrayList<String> responses;

    /**
     * Crear un generador de respuestas.
     */
    public Responder()
    {
        randomGenerator = new Random();
        responses = new ArrayList<String>();
        fillResponses();
    }

    /**
     * Generar una respuesta.
     * @return  Una cadena que debe mostrarse como
     *          respuesta
     */
    public String generateResponse()
    {
        // Seleccionar un número aleatorio para el índice de
        // la lista predeterminada de respuestas. El número
        // estará entre 0 (inclusive) y el tamaño
        // de la lista (exclusive).
        int index = randomGenerator.nextInt(responses.size());
        return responses.get(index);
    }

    /**
     * Construir una lista de respuestas predeterminadas,
     * a partir de la cual podamos seleccionar una si
     * no sabemos qué otra cosa decir.
     */
```

Código 5.3
(*continuación*)
El código fuente de `Responder` con respuestas aleatorias.

```
    private void fillResponses()
    {
        responses.add("That sounds odd. Could you describe \n" +
                      "that problem in more detail?");
        responses.add("No other customer has ever \n" +
                      "complained about this before. \n" +
                      "What is your system configuration?");
        responses.add("That's a known problem with Vista." +
                      "Windows 7 is much better.");
        responses.add("I need a bit more information on that.");
        responses.add("Have you checked that you do not \n" +
                      "have a dll conflict?");
        responses.add("That is explained in the manual. \n" +
                      "Have you read the manual?");
        responses.add("Your description is a bit \n" +
                      "wishy-washy. Have you got an expert \n" +
                      "there with you who could describe \n" +
                      "this more precisely?");
        responses.add("That's not a bug, it's a feature!");
        responses.add("Could you elaborate on that?");
    }
}
```

En esta versión, hemos puesto el código que rellena la lista de respuestas en su propio método, denominado `fillResponses`, que se invoca desde el constructor. Esto garantiza que la lista de respuestas sea rellenada en cuanto se cree un objeto `Responder`, pero el código fuente tras rellenar la lista se mantiene separado con el fin de que la clase sea capaz de leer y comprender.

El segmento de código más interesante en esta clase se encuentra en el método `generateResponse`. Quitando los comentarios, el código es el siguiente:

```
public String generateResponse()
{
    int index = randomGenerator.nextInt(responses.size());
    return responses.get(index);
}
```

La primera línea del código en este método hace tres cosas:

- Calcula el tamaño de la lista de respuestas llamando a su método `size`.
- Genera un número aleatorio comprendido entre 0 (inclusive) y `size` (exclusive).
- Almacena ese número aleatorio en la variable local `index`.

Si le parece que esto es demasiado código para una sola línea, también podría escribir:

```
int listSize = responses.size();
int index = randomGenerator.nextInt(listSize);
```

Este código es equivalente a la primera línea mostrada anteriormente. De nuevo, la versión preferible será la que a cada uno le resulte más fácil de leer.

Es importante observar que este segmento de código generará un número aleatorio en el rango de 0 a listSize-1 (inclusive). Esto encaja perfectamente con los índices legales para un ArrayList. Recuerde que el rango de índices para un ArrayList de tamaño listSize va de 0 a listSize-1. Por tanto, el número aleatorio calculado nos da un índice perfecto para acceder aleatoriamente a uno de los elementos de la lista completa.

La última línea del método es

```
return responses.get(index);
```

Esta línea hace dos cosas:

- Extrae la respuesta situada en la posición index utilizando el método get.
- Devuelve la cadena seleccionada como resultado del método, utilizando la instrucción return.

Si no tiene cuidado, su código podría generar un número aleatorio que quede fuera del rango de índices válidos del objeto ArrayList. Cuando luego intente utilizarlo como índice para acceder a un elemento de la lista, obtendrá un error IndexOutOfBoundsException.

5.4.4 Lectura de la documentación de las clases parametrizadas

Hasta ahora, le hemos pedido que examine la documentación de la clase String del paquete java.lang y de la clase Random del paquete java.util. Puede que haya observado al hacer esto que algunos nombres de clases en la lista contenida en la documentación tienen un aspecto ligeramente distinto, como por ejemplo ArrayList<E> o HashMap<K, V>. Es decir, el nombre de la clase va seguido por una cierta información adicional que aparece entre corchetes angulares. Las clases de este estilo se denominan *clases parametrizadas* o *clases genéricas*. La información encerrada en los corchetes angulares nos dice que al utilizar estas clases debemos suministrar uno o más nombres de tipo entre corchetes angulares para completar la definición. Ya hemos visto aplicada esta idea en el Capítulo 4, donde hemos utilizado ArrayList parametrizándola con nombres de tipo como String. También pueden parametrizarse con cualquier otro tipo:

```
private ArrayList<String> notes;
private ArrayList<Student> students;
```

Puesto que podemos parametrizar un ArrayList con cualquier otro tipo de clase que elijamos, este hecho se refleja en la documentación de la API. Por tanto, si examinamos la lista de métodos de ArrayList<E>, podremos ver métodos como:

```
boolean add(E o)
E get(int index)
```

Esto nos dice que el tipo de objetos que podemos añadir a un ArrayList (con add) depende del tipo utilizado para parametrizarla y que el tipo de los objetos devueltos por su método get depende de la misma manera de ese tipo empleado en la parametrización. De hecho, si creamos un objeto ArrayList<String>, lo que la documentación nos dice es que el objeto tiene los siguientes dos métodos:

```
boolean add(String o)
String get(int index)
```

mientras que si creamos un objeto ArrayList<Student>, entonces tendrá los otros dos métodos:

```
boolean add(Student o)
Student get(int index)
```

En secciones posteriores de este capítulo le pediremos que examine la documentación para ver otros tipos parametrizados.

5.5 Paquetes e importación

Hay todavía dos líneas al principio del código fuente que tenemos que comentar:

```
import java.util.ArrayList;
import java.util.Random;
```

Nos hemos encontrado con la instrucción de importación (`import`) por primera ver en el Capítulo 4. Ahora es el momento de examinarla con más detalle.

Las clases Java que están almacenadas en la librería de clases no están disponibles automáticamente para ser utilizadas, como sí lo están las otras clases del proyecto actual. En lugar de ello, debemos indicar en nuestro código fuente que nos gustaría utilizar una clase de la librería. Esto se denomina *importar* la clase y se hace mediante la instrucción `import`. La instrucción `import` tiene el formato

```
import nombre-clase-cualificado;
```

Puesto que la librería Java contiene varios miles de clases, hace falta una cierta estructura en la organización de la librería para facilitar el manejo de ese gran número de clases. Java utiliza *paquetes* para clasificar las clases de librería en grupos de clases relacionadas. Los paquetes están anidados (es decir, los paquetes pueden contener otros paquetes).

Las clases `ArrayList` y `Random` se encuentran ambas en el paquete `java.util`. Esta información puede encontrarse en la documentación de la clase. El *nombre completo* o *nombre cualificado* de una clase es el nombre de su paquete, seguido por un punto y por el nombre de la clase. Por tanto, los nombres cualificados de las dos clases que hemos usado aquí son: `java.util.ArrayList` y `java.util.Random`.

Java también nos permite importar paquetes completos con instrucciones de la forma

```
import nombre-paquete.*;
```

Por tanto, la siguiente instrucción importaría todos los nombres de clase del paquete `java.util`:

```
import java.util.*;
```

Enumerar por separado todas las clases utilizadas, como en nuestra primera versión, requiere algo más de trabajo de escritura, pero resulta adecuado desde el punto de vista de la documentación. Indica claramente qué clases están siendo utilizadas realmente por nuestra clase. Por tanto, en este libro, tenderemos a utilizar el estilo del primer ejemplo, enumerando por separado todas las clases importadas.

Existe una excepción a estas reglas: algunas clases se emplean tan frecuentemente que casi todas las demás clases tendrán que importarlas. Estas clases se han incluido en el paquete `java.lang`, y este paquete se importa de manera automática en todas las clases. Por tanto, no necesitamos escribir instrucciones de importación para las clases contenidas en `java.lang`. La clase `String` es un ejemplo de ese tipo de clases.

Ejercicio 5.21 Implemente en su versión del sistema *tech-support* la solución de respuesta aleatoria que hemos analizado aquí.

Ejercicio 5.22 ¿Qué sucede cuando se añaden más (o menos) posibles respuestas a la lista de respuestas existente? ¿Seguirá funcionando adecuadamente el mecanismo de selección de una respuesta aleatoria? ¿Por qué?

La solución explicada aquí se encuentra en los proyectos del libro con el nombre de *tech-support2*. Sin embargo, le recomendamos que la implemente usted por sí mismo como extensión de la versión base.

5.6 Utilización de mapas para asociaciones

Ahora tenemos una solución para nuestro sistema de soporte técnico capaz de generar respuestas aleatorias. Esta es mejor que la primera versión, pero sigue sin ser demasiado convincente. En particular, la entrada del usuario no influye de ninguna forma sobre la respuesta. Esta es precisamente el área que queremos mejorar ahora.

El plan consistirá en disponer de un conjunto de palabras que tienen una alta probabilidad de aparecer en las preguntas típicas, y asociar esas palabras con respuestas concretas. Si la entrada del usuario contiene una de nuestras palabras predefinidas, podremos generar una respuesta relacionada. Este continúa siendo un método muy burdo, porque no captura nada del significado implícito en la entrada del usuario, ni tampoco reconoce un contexto, pero puede resultar sorprendentemente efectivo. Y, sobre todo, se trata de un buen paso con el que seguir mejorando nuestro programa.

Para hacer esto, utilizaremos un `HashMap`. Puede encontrar la documentación de la clase `HashMap` en la documentación de la librería Java. `HashMap` es una especialización de `Map`, que también está documentada. Se encontrará con que tiene que leer la documentación de ambas clases para comprender lo que es un `HashMap` y cómo funciona.

Ejercicio 5.23 ¿Qué es un `HashMap`? ¿Cuál es su propósito y cómo se usa? Responda a estas cuestiones por escrito y utilice la documentación de la librería Java para `Map` y `HashMap` con el fin de preparar sus respuestas. Tenga en cuenta que le resultará difícil comprender todos los conceptos, ya que la documentación de estas clases no es muy buena. Explicaremos los detalles más adelante en el capítulo, pero trate de averiguar por su cuenta todo lo que pueda antes de continuar leyendo.

Ejercicio 5.24 `HashMap` es una clase parametrizada. Enumere aquellos de sus métodos que dependen de los tipos utilizados para parametrizarla. ¿Cree que podría utilizarse el mismo tipo para sus parámetros?

5.6.1 El concepto de mapa

Un mapa es una colección de parejas clave/valor de objetos. Como con `ArrayList`, un mapa puede almacenar un tipo flexible de entradas. Una diferencia entre `ArrayList` y `Map` es que con `Map` cada entrada no es un objeto, sino una *pareja* de objetos. Esta pareja está formada por objeto *clave* y un objeto *valor*.

Concepto:

Un **mapa** es una colección que almacena parejas clave/valor como entradas. Se pueden buscar valores proporcionando la clave.

En lugar de buscar entradas en esta colección utilizando un índice entero (como hicimos con `ArrayList`), utilizamos el objeto clave para buscar el objeto valor.

Un ejemplo de mapa en la vida cotidiana sería una guía telefónica. La guía telefónica contiene entradas y cada entrada es una pareja: un nombre y un número de teléfono. Utilizamos la guía telefónica buscando un nombre y leyendo el número de teléfono asociado. No empleamos ningún índice (la posición de la entrada en la guía) para averiguar el número de teléfono.

Un mapa se puede organizar de tal manera que sea fácil buscar el valor correspondiente a una clave. En el caso de la guía telefónica, esto se hace mediante la ordenación alfabética. Almacenando las entradas en orden alfabético de sus claves, resulta sencillo localizar la clave y consultar el valor asociado. La búsqueda inversa (localizar la clave correspondiente a un valor –es decir, localizar el nombre para un número de teléfono dado) no es tan sencilla con un mapa. Al igual que con una guía telefónica, es posible realizar una búsqueda inversa en un mapa, pero se necesita un tiempo relativamente largo. Por tanto, los mapas son ideales para búsquedas en una sola dirección, en las que conocemos la clave de búsqueda y necesitamos saber el valor asociado con esa clave.

5.6.2 Utilización de un HashMap

`HashMap` es una implementación específica de `Map`. Los métodos más importantes de la clase `HashMap` son `put` y `get`.

El método `put` inserta una entrada en el mapa, mientras que `get` extrae el valor correspondiente a una clave especificada. El siguiente fragmento de código crea un `HashMap` y inserta en él tres entradas. Cada entrada es una pareja clave/valor compuesta por un nombre y un número de teléfono.

```
HashMap<String, String> phoneBook = new HashMap<String, String>();

phoneBook.put("Charles Nguyen", "(531) 9392 4587");
phoneBook.put("Lisa Jones", "(402) 4536 4674");
phoneBook.put("William H. Smith", "(998) 5488 0123");
```

Como hemos visto con `ArrayList`, al declarar una variable `HashMap` y al crear un objeto `HashMap`, tenemos que indicar qué tipo de objetos se almacenarán en el mapa y, adicionalmente, qué tipo de objetos se emplearán como clave. Para la guía de teléfonos utilizaríamos cadenas de caracteres tanto para las claves como para los valores, pero en otros casos ambos tipos serán diferentes.

Como hemos visto en la Sección 4.4.2, si estamos utilizando Java 7 (o una versión más reciente), la especificación de tipo genérico del lado derecho de la asignación puede omitirse, como en la siguiente instrucción:

```
HashMap<String, String> phoneBook = new HashMap<>();
```

Esto se conoce con el nombre de "operador diamante" debido a los dos corchetes angulares vacíos: <>.

En Java 7, esta línea es equivalente a la primera línea anterior (pero en Java 6 no se podría compilar). Si se omiten los parámetros genéricos, el compilador se limitará a asumir los mismos parámetros usados en el lado izquierdo de la asignación. Por tanto, el efecto es exactamente el mismo, simplemente nos ahorramos escribir unos cuantos caracteres.

El siguiente código encuentra el número de teléfono de Lisa Jones y lo imprime.

```
String number = phoneBook.get("Lisa Jones");
System.out.println(number);
```

Observe que pasamos la clave (el nombre "Lisa Jones") al método `get` para recibir el valor (el número de teléfono).

Consulte de nuevo la documentación de los métodos `get` y `put` de la clase `HashMap` y vea si las explicaciones se corresponden con lo que ha podido entender acerca de los métodos de esta clase.

Ejercicio 5.25 ¿Cómo podemos comprobar cuántas entradas hay almacenadas en un mapa?

Ejercicio 5.26 Cree una clase `MapTester` (en su proyecto actual o en un nuevo proyecto). En ella, utilice un `HashMap` para implementar una guía telefónica similar a la del ejemplo anterior. Recuerde que tiene que importar `java.util.HashMap`. En esta clase, implemente dos métodos:

```
public void enterNumber(String name, String number)
```

y

```
public String lookupNumber(String name)
```

Estos métodos deben utilizar los métodos `put` y `get` de la clase `HashMap` para implementar su funcionalidad, consistente en introducir y leer un número de teléfono, respectivamente.

Ejercicio 5.27 ¿Qué sucede cuando añadimos una entrada a un mapa con una clave que ya existe en ese mapa?

Ejercicio 5.28 ¿Qué sucede cuando añadimos una entrada a un mapa con dos claves diferentes?

Ejercicio 5.29 ¿Cómo podemos comprobar si una clave determinada está contenida en un cierto mapa? (Proporcione un ejemplo de código Java).

Ejercicio 5.30 ¿Qué sucede cuando tratamos de buscar un valor y la clave no existe en el mapa?

Ejercicio 5.31 ¿Cómo comprobamos cuántas entradas hay en un mapa?

Ejercicio 5.32 ¿Cómo pueden imprimirse todas las claves almacenadas actualmente en un mapa?

5.6.3 Utilización de un mapa para el sistema TechSupport

En el sistema TechSupport, podemos hacer un buen uso de los mapas utilizando como claves una serie de palabras conocidas y empleando como valores las respuestas asociadas. El Código 5.4 muestra un ejemplo en el que se crea un `HashMap` denominado `responseMap` y se introducen en él tres entradas. Por ejemplo, la palabra "slow" (lento) está asociada con el texto:

*"I think this has to do with your hardware. Upgrading your processor should solve all performance problems. Have you got a problem with our software?"**

---

* *"Creo que esto tiene que ver con su hardware. Actualizar el procesador debería resolver todos los problemas de rendimiento. ¿Ha experimentado algún problema con nuestro software?*

Código 5.4
Asociación de palabras seleccionadas con posibles respuestas.

```
private HashMap<String, String> responseMap;
...
public Responder()
{
    responseMap = new HashMap<String, String>();
    fillResponseMap();
}

/**
 * Introducir en el mapa de respuestas todas las palabras clave
 * conocidas y sus respuestas asociadas.
 */

private void fillResponseMap()
{
    responseMap.put("slow",
        "I think this has to do with your hardware. \n" +
        "Upgrading your processor should solve all " +
        "performance problems. \n" +
        "Have you got a problem with our software?");
    responseMap.put("bug",
        "Well, you know, all software has some bugs. \n" +
        "But our software engineers are working very " +
        "hard to fix them. \n" +
        "Can you describe the problem a bit further?");
    responseMap.put("expensive",
        "The cost of our product is quite competitive. \n" +
        "Have you looked around and " +
        "really compared our features?");
}
```

Ahora, cuando alguien plantee una cuestión que contenga la palabra *slow*, podemos buscar e imprimir esta respuesta. Observe que la cadena de respuesta en el código fuente abarca varias líneas, pero está concatenada con el operador + , por lo que en el `HashMap` se introduce como valor una única cadena de caracteres.

Un primer intento de escribir un método para generar las respuestas podría, con esto, tener el aspecto del método `generateResponse` que se proporciona a continuación. Aquí, para simplificar las cosas por el momento, suponemos que el usuario introduce una única palabra (por ejemplo, *slow*).

```
public String generateResponse(String word)
{
    String response = responseMap.get(word);
    if(response != null) {
        return response;
    }
```

```
        else {
            // Si llegamos aquí es que la palabra no ha sido reconocida.
            // En ese caso, seleccionamos una de nuestras respuestas
            // predeterminadas.
            return pickDefaultResponse();
        }
    }
```

En este fragmento de código, buscamos en nuestro mapa de respuestas la palabra introducida por el usuario. Si encontramos una entrada, la utilizamos como respuesta. Si no encontramos una entrada para esa palabra, llamamos al método `pickDefaultResponse`. Este método puede contener ahora el código de la versión anterior de nuestro método `generateResponse`, que selecciona aleatoriamente una de las respuestas predeterminadas (como se muestra en el Código 5.3). La nueva lógica consiste, por tanto, en seleccionar una respuesta apropiada si reconocemos una palabra, o devolver en caso contrario una respuesta aleatoria, seleccionada de entre nuestra lista de respuestas predeterminadas.

Ejercicio 5.33 Implemente los cambios expuestos aquí en su propia versión del sistema TechSupport. Pruébela para ver hasta qué punto resulta más natural que la versión anterior.

Esta técnica de asociar palabras clave con respuestas funciona bastante bien, siempre y cuando el usuario no introduzca preguntas completas, sino solo palabras sueltas. La mejora final para completar la aplicación consiste en permitir al usuario introducir preguntas completas y luego seleccionar las respuestas asociadas si reconocemos dentro de esas preguntas algunas de las palabras clave.

Esto plantea el problema de reconocer esas palabras clave dentro de la frase introducida por el usuario. En la versión actual, la entrada del usuario es devuelta por `InputReader` como una única cadena de caracteres. Vamos a cambiar ahora esto a una nueva versión, en la que `InputReader` devolverá la entrada como un conjunto de palabras. Técnicamente, se tratará de un conjunto de cadenas de caracteres, en el que cada cadena del conjunto representará una única palabra introducida por el usuario.

Si podemos hacer esto, entonces podemos pasar el conjunto completo a `Responder`, que podrá a su vez comprobar cada palabra del conjunto, para ver si es una de las palabras clave conocidas y dispone de una respuesta asociada.

Para conseguir esto en Java, necesitamos más información acerca de dos cosas: cómo partir en palabras una única cadena de caracteres que contiene una frase completa y cómo utilizar conjuntos. Estas dos cuestiones se analizan en las siguientes dos secciones.

5.7 Utilización de conjuntos

La librería estándar Java incluye diferentes variantes de conjuntos implementados en clases distintas. La clase que vamos a utilizar aquí se denomina `HashSet`.

Ejercicio 5.34 ¿Cuáles son las similitudes y diferencias entre un `HashSet` y un `ArrayList`? Utilice las descripciones de `Set`, `HashSet`, `List` y `ArrayList` disponibles en la documentación de la librería, porque `HashSet` es un caso especial de `Set` y `ArrayList` es un caso especial de `List`.

Los dos tipos de funcionalidad que necesitamos son la capacidad de introducir elementos en el conjunto y de extraer esos elementos posteriormente. Afortunadamente, estas tareas apenas contienen nada que nos resulte nuevo. Considere el siguiente fragmento de código:

```
import java.util.HashSet;
...
HashSet<String> mySet = new HashSet<String>();

mySet.add("one");
mySet.add("two");
mySet.add("three");
```

Compare este código con las instrucciones necesarias para introducir elementos en un `ArrayList`. Apenas hay diferencia salvo porque en esta ocasión creamos un `HashSet` en lugar de un `ArrayList`. Veamos cómo se iteraría a través de los elementos:

```
for(String item : mySet) {
    hacer algo con ese elemento
}
```

Concepto:

Un **conjunto** es una colección que almacena cada elemento individual como máximo una vez. No mantiene ningún orden específico.

De nuevo, estas instrucciones son iguales a las que hemos utilizado para iterar a través de un `ArrayList` en el Capítulo 4.

En resumen, la utilización de colecciones en Java es bastante similar para los distintos tipos de colección. Una vez que se comprende cómo utilizar una de ellas, se pueden usar todas. Las diferencias radican, realmente, en el comportamiento de cada colección. Por ejemplo, una lista mantendrá todos los elementos que se introduzcan en el orden deseado, proporcionará acceso a esos elementos mediante un índice y puede contener el mismo elemento varias veces. Un conjunto, por el contrario, no mantiene ningún orden específico (los elementos pueden ser devueltos en un bucle for-each en un orden distinto de aquel en el que fueron introducidos) y garantiza que cada elemento se introduzca en el conjunto como máximo una vez. Introducir un elemento una segunda vez simplemente no tiene ningún efecto.

List, Map y Set Resulta tentador asumir que un `HashSet` debe emplearse de una forma similar a un `HashMap`. De hecho, como hemos ilustrado, la utilización de un `HashSet` se parece mucho más al uso de un `ArrayList`.

A la hora de intentar entender cómo se utilizan los distintos tipos de clases de colección, resulta útil prestar una especial atención a sus nombres. Los nombres están compuestos de dos partes, como por ejemplo: "Array" "List." La segunda mitad nos dice qué tipo de colección se trata (List, Map, Set), mientras que el primero nos informa acerca de cómo está implementada (por ejemplo, utilizando una matriz –*array* en inglés).

Para utilizar colecciones, el tipo de la colección (la segunda parte) es el más importante. Hemos explicado anteriormente que a menudo podemos abstraernos de la implementación; no necesitamos pensar demasiado en ella. Por tanto, para nuestros propósitos, un `HashSet` y un `TreeSet` son muy similares. Ambos son conjuntos, por lo que se comportan de la misma manera. La diferencia solo radica en su implementación, que únicamente es importante cuando comenzamos a pensar en términos de eficiencia: un tipo de implementación llevará a cabo ciertas operaciones mucho más rápido que otro. Sin embargo, las consideraciones de eficiencia entran en escena mucho más tarde y solo cuando tengamos colecciones de muy gran tamaño o aplicaciones en las que el rendimiento sea crítico.

5.8 División de cadenas de caracteres

Ahora que hemos visto cómo emplear un conjunto, podemos investigar cómo se puede dividir la cadena de entrada en palabras separadas, para almacenarlas en un conjunto de palabras. La solución se muestra en una nueva versión del método `getInput` de `InputReader` (Código 5.5).

Código 5.5
El método `getInput` que devuelve un conjunto de palabras.

```
/**
 * Leer una línea de texto de la entrada estándar (el terminal
 * de texto) y devolverla en forma de un conjunto de palabras.
 *
 * @return  Un conjunto de objetos String, donde cada String es una
 *          de las palabras tecleadas por el usuario.
 */
public HashSet<String> getInput()
{
    System.out.print("> ");          // imprimir el indicativo
    String inputLine = reader.nextLine().trim().toLowerCase();
    String[] wordArray = inputLine.split(" ");

    // Añadir palabras del array al hashset
    HashSet<String> words = new HashSet<String>();
    for(String word : wordArray) {
        words.add(word);
    }

    return words;
}
```

Aquí, además de utilizar un `HashSet`, usamos también el método `split`, que es un método estándar de la clase `String`.

El método `split` puede dividir una cadena en una serie de subcadenas separadas y devolverlas en una matriz de cadenas. El parámetro del método `split` define cuál es el tipo de caracteres según los cuales hay que dividir la cadena original. Lo que hemos hecho es definir que queremos cortar nuestra cadena por cada carácter de espaciado.

Las siguientes líneas de código crean un `HashSet` y copian las palabras de la matriz en el conjunto, antes de devolver ese conjunto.[3]

---

[3] Hay otra forma más corta y elegante de hacer esto. Podríamos escribir

```
HashSet<String> words = new HashSet<String>(Arrays.asList(wordArray));
```

para sustituir nuestras cuatro líneas de código. Esto utiliza la clase `Arrays` de la librería estándar y un *método estático* (también denominado *método de clase*) que por el momento no vamos a analizar. Si tiene curiosidad, puede obtener más información sobre los *métodos de clase* en la Sección 6.15 e intentar utilizarlo en esta versión.

Ejercicio 5.35 El método `split` es más potente de lo que puede deducirse a simple vista de nuestro ejemplo. ¿Cómo podemos definir exactamente la forma en que hay que dividir una cadena? Proporcione algunos ejemplos.

Ejercicio 5.36 ¿Cómo invocaría el método `split` si quisiera dividir una cadena atendiendo a los caracteres de espaciado o tabulación? ¿Cómo podría dividir una cadena en la que las palabras estuvieran separadas por caracteres de dos puntos (:)?

Ejercicio 5.37 ¿Cuál es la diferencia en el resultado de devolver las palabras en un `HashSet`, si lo comparamos con devolverlas en un `ArrayList`?

Ejercicio 5.38 ¿Qué sucede si hay más de un espacio entre dos palabras (por ejemplo, dos o tres espacios? ¿Genera esto algún problema?

Ejercicio 5.39 *Ejercicio avanzado* Lea la nota al pie acerca del método `Arrays.asList`. Localice y lea las secciones de este libro que se ocupan de las variables de clase y de los métodos de clase. Explique con sus propias palabras cómo funciona esto.

Cite otros ejemplos de métodos que proporcione la clase `Arrays`.

Cree una clase denominada `SortingTest`. En ella, cree un método que acepte como parámetro una matriz de valores `int` e imprima en el terminal los elementos de forma ordenada (comenzando por el de menor valor).

5.9 Finalización del sistema *TechSupport*

Para juntar todos los conceptos, también tenemos que ajustar las clases `SupportSystem` y `Responder` para tratar correctamente con un conjunto de palabras en lugar de con una única cadena de caracteres. El Código 5.6 muestra la nueva versión del método `start` de la clase `SupportSystem`. Los cambios no son demasiados:

- La variable `input` que recibe el resultado de `reader.getInput()` es ahora de tipo `HashSet`.
- La comprobación para la finalización del programa se realiza empleando el método `contains` de la clase `HashSet`, en lugar de un método `String`. (Busque este método en la documentación.)
- Hay que importar la clase `HashSet` utilizando una instrucción `import` (que no se muestra aquí).

Finalmente, tenemos que ampliar el método `generateResponse` de la clase `Responder` para que acepte un conjunto de palabras como parámetro. Después, el método tendrá que iterar a través de esas palabras y contrastar cada una de ellas con nuestro mapa de palabras conocidas. Si se reconoce cualquiera de las palabras, se devuelve inmediatamente la respuesta asociada. Si no se reconoce ninguna de las palabras, como antes, seleccionaremos una de las respuestas predeterminadas. El Código 5.7 muestra la solución.

Código 5.6
Versión final del método `start`.

```
public void start()
{
    boolean finished = false;

    printWelcome();

    while(!finished) {
        HashSet<String> input = reader.getInput();

        if(input.contains("bye")) {
            finished = true;
        }
        else {
            String response = responder.generateResponse(input);
            System.out.println(response);
        }
    }
    printGoodbye();
}
```

Código 5.7
Versión final del método `generateResponse`.

```
public String generateResponse(HashSet<String> words)
{
    for(String word : words) {
        String response = responseMap.get(word);
        if(response != null) {
            return response;
        }
    }

    // Si llegamos aquí es que la palabra no ha sido reconocida.
    // En ese caso, seleccionamos una de nuestras respuestas
    // predeterminadas.
    return pickDefaultResponse();
}
```

Este es el último cambio que vamos a realizar en este capítulo a la aplicación que venimos analizando. La solución presentada en el proyecto *tech-support-complete* contiene todos estos cambios. También contiene más asociaciones de palabras con respuestas de las que se han mostrado aquí.

Se podrían hacer muchas mejoras en esta aplicación, pero no vamos a analizarlas aquí. En lugar de ello, le sugerimos unas cuantas en forma de ejercicios, que se dejan para que el lector practique. Algunos de estos son ejercicios de programación realmente difíciles.

Ejercicio 5.40 Implemente los cambios finales que hemos explicado anteriormente en su propia versión del programa.

Ejercicio 5.41 Añada más asociaciones entre palabras/respuestas a su aplicación. Puede copiar algunas de las soluciones proporcionadas y añadir otras de su propia cosecha.

Ejercicio 5.42 Modifique el programa para asegurarse de que nunca se repita dos veces seguidas la misma respuesta predeterminada.

Ejercicio 5.43 En ocasiones, dos palabras (o variantes de una palabra) tienen asignada la misma respuesta. Incluya esta solución en su programa asignando sinónimos o expresiones relacionadas a la misma cadena de caracteres, de modo que no hagan falta múltiples entradas en el mapa de respuestas para la misma respuesta.

Ejercicio 5.44 Identifique múltiples palabras clave dentro de la entrada del usuario y genere en ese caso una respuesta más apropiada.

Ejercicio 5.45 Cuando no reconozca ninguna palabra, utilice otras palabras de la entrada del usuario para seleccionar una respuesta predeterminada más ajustada: por ejemplo, palabras como *why* (por qué), *how* (cómo) y *who* (quién).

5.10 Escritura de la documentación de las clases

Cuando trabaje con sus propios proyectos, es importante que escriba la documentación de sus clases a medida que desarrolle el código fuente. Es bastante común que los programadores no se tomen suficientemente en serio la documentación, y esto crea, muy frecuentemente, graves problemas más adelante.

Si no suministra la suficiente documentación, le puede resultar muy difícil a otro programador (¡o a usted mismo más adelante!) comprender sus clases. Normalmente, lo que hay que hacer en ese caso es leer la implementación de la clase y tratar de imaginarse qué es lo que hace. Esto puede funcionar con un pequeño proyecto estudiantil, pero crea serios problemas en los proyectos de desarrollo del mundo real.

Concepto:

La **documentación** de una clase debe ser lo suficientemente detallada como para que otros programadores puedan utilizar la clase sin necesidad de leer su implementación.

Es frecuente que las aplicaciones comerciales estén compuestas por centenares de miles de líneas de código agrupadas en varios miles de clases. ¡Imagine tener que leer todo eso para poder comprender cómo funciona una aplicación! Jamás podríamos hacerlo.

Cuando hemos utilizado clases de la librería Java, tales como `HashSet` o `Random`, lo único que hemos necesitado es la documentación para averiguar como utilizarlas. Nunca hemos mirado la implementación de esas clases. Este enfoque funciona porque esas clases estaban lo suficientemente bien documentadas (aun cuando incluso esa documentación podría mejorarse). Nuestra tarea habría sido mucho más difícil si se nos hubiera exigido leer la implementación de las clases antes de usarlas.

En un equipo de desarrollo de software, la tarea de implementación de clases suele ser compartida entre varios programadores. Mientras que usted podría ser el responsable de implementar la clase `SupportSystem` de nuestro último ejemplo, algún otro programador podría implementar la clase `InputReader`. Por tanto, puede que a usted le toque escribir una clase a la vez que realiza llamadas a los métodos de otras clases.

El mismo argumento utilizado para las clases de librería sigue siendo válido para las clases que nosotros escribamos: si podemos emplear las clases sin tener que leer y comprender la implementación completa, nuestra tarea será mucho más sencilla. Al igual que sucede con las clases de librería, lo único que necesitamos conocer es la interfaz pública de la clase, en lugar de la implementación. Por tanto, es importante escribir también una buena documentación para nuestras clases.

Los sistemas Java incluyen una herramientas denominada `javadoc` que puede utilizarse para generar la descripción de esas interfaces a partir del código fuente. La documentación de la librería estándar que hemos utilizado, por ejemplo, fue creada a partir del código fuente de las clases mediante `javadoc`.

5.10.1 Utilización de javadoc en BlueJ

El entorno BlueJ utiliza `javadoc` para permitirnos crear documentación para nuestras clases de dos formas distintas:

- Podemos ver la documentación para una única clase pasando el selector emergente situado en la parte superior derecha de la ventana del editor de *Source Code* a *Documentation*, o seleccionando *Toggle Documentation View* (Cambiar a vista de documentación) en el menú *Tools* (Herramientas) del editor.
- Podemos usar la función *Project Documentation* disponible en el menú *Tools* de la ventana principal para generar la documentación correspondiente a todas las clases del proyecto.

El tutorial de BlueJ proporciona más detalles en caso de que esté interesado. Puede encontrar el tutorial de BlueJ en el menú *Help* de BlueJ.

5.10.2 Elementos de la documentación de una clase

La documentación de una clase debe incluir al menos:

- El nombre de la clase.
- Un comentario que describa el propósito global y las características de la clase.
- Un número de versión.
- El nombre del autor (o autores).
- La documentación para cada constructor y cada método.

La documentación de cada constructor y cada método debe incluir:

- El nombre del método.
- El tipo de retorno.
- Los nombres y tipos de los parámetros.
- Una descripción del propósito y función del método.
- Una descripción de cada parámetro.
- Una descripción del valor devuelto.

Además, cada proyecto completo debe tener un comentario global del proyecto, que a menudo estará contenido en un archivo "ReadMe" (Léame). En BlueJ, se puede acceder a este comentario del proyecto a través de la nota de texto mostrada en la esquina superior izquierda del diagrama de clases.

Ejercicio 5.46 Utilice la función *Project Documentation* de BlueJ para generar la documentación del proyecto TechSupport. Examínela. ¿Es precisa? ¿Es completa? ¿Qué partes son útiles? ¿Cuáles no lo son? ¿Ha podido encontrar algún error en la documentación?

Algunos elementos de la documentación, como los nombres y parámetros de los métodos, siempre pueden extraerse del código fuente. Otras partes, como los comentarios que describen las clases, los métodos y los parámetros, necesitan más atención, ya que podemos olvidarnos fácilmente de ellos o pueden ser incompletos o incorrectos.

En Java, los comentarios `javadoc` se escriben con un símbolo especial de comentario al principio:

```
/**
   Este es un comentario javadoc.
 */
```

El símbolo de inicio del comentario tiene que tener dos asteriscos para que sea reconocido como un comentario `javadoc`. Dicho comentario, si precede inmediatamente a la declaración de la clase se lee como un comentario de la clase. Si el comentario está justo encima de la signatura de un método se le considera un comentario del método.

Los detalles exactos de cómo se produce y formatea la documentación son diferentes en los distintos lenguajes y entornos de programación. Sin embargo, el contenido debe ser más o menos siempre el mismo.

En Java, utilizando `javadoc`, hay disponibles varios símbolos clave especiales para dar formato a la documentación. Estos símbolos clave comienzan con el símbolo `@` e incluyen:

```
@version
@author
@param
@return
```

Ejercicio 5.47 Localice ejemplos de símbolos clave `javadoc` en el código fuente del proyecto *TechSupport*. ¿Cómo influyen en el formateo de la documentación?

Ejercicio 5.48 Consiga información acerca de otros símbolos clave `javadoc` y descríbalos. Un lugar en el que puede mirar es en la documentación en línea de la distribución Java de Oracle. Contiene un documento denominado *javadoc – The Java API Documentation Generator* (por ejemplo, en `http://download.oracle.com/javase/6/docs/technotes/tools/windows/javadoc.html`). En este documento, los símbolos clave se denominan *javadoc tags* (etiquetas javadoc).

Ejercicio 5.49 Documente apropiadamente todas las clases de su versión del proyecto *TechSupport*.

5.11 Public y private

Es el momento de explicar con más detalle un aspecto de las clases con el que nos hemos encontrado ya varias veces pero acerca del cual no hemos hablado: los *modificadores de acceso.*

Los modificadores de acceso son las palabras clave `public` o `private` situadas al principio de las declaraciones de campo y de las signaturas de método. Por ejemplo:

```
// declaración de campo
private int numberOfSeats;

// métodos
public void setAge(int replacementAge)
{ ...
}

private int computeAverage()
{ ...
}
```

Los campos, los constructores y los métodos pueden ser públicos o privados, aunque hasta ahora hemos visto campos privados y constructores y métodos públicos. Volveremos sobre esto más adelante.

Concepto:

Los **modificadores de acceso** definen la visibilidad de un campo, constructor, o método. Los elementos públicos son accesibles desde dentro de la misma clase y desde otras clases; los elementos privados solo son accesibles desde dentro de la misma clase.

Los modificadores de acceso definen la visibilidad de un campo, un constructor o un método. Por ejemplo, si un método es público, se puede invocar desde la misma clase o desde cualquier otra clase. Por el contrario, los métodos privados solo pueden invocarse desde la misma clase en la que están declarados. No son visibles para otras clases.

Ahora que hemos hablado de la diferencia entre la interfaz y la implementación de una clase (Sección 5.3.1), podemos entender más fácilmente el propósito de estas palabras clave.

Recuerde: la interfaz de una clase es el conjunto de detalles que necesita ver cualquier otro programador que use esa clase. Proporciona información acerca de cómo utilizar la clase. La interfaz incluye las signaturas de los constructores y métodos, además de una serie de comentarios. También se la denomina parte *pública* de una clase. Su propósito es definir *lo que la clase hace.*

La implementación es la sección de una clase que define precisamente *cómo* funciona la clase. Los cuerpos de los métodos, que contienen las instrucciones Java y la mayoría de los campos son parte de la implementación. La implementación también se denomina parte *privada* de una clase. El usuario de una clase no necesita conocer su implementación. De hecho, hay buenas razones por las que a un usuario *debería impedírsele conocer* la implementación (o al menos hacer uso de dicho conocimiento). Este principio se conoce como *ocultamiento de la información.*

La palabra clave `public` declara un elemento de una clase (un campo o un método) como parte de la interfaz (es decir, que es públicamente visible); la palabra clave `private` declara que es parte de la implementación (es decir, que está oculto frente a posibles accesos externos).

5.11.1 Ocultamiento de la información

En muchos lenguajes de programación orientados a objetos, las interioridades de una clase –su implementación– se ocultan a ojos de otras clases. Hay dos caras en este ocultamiento. En primer lugar, un programador que haga uso de una clase, *no debería necesitar conocer* sus interioridades; en segundo lugar, a un usuario *no debería permitírsele conocer* esas interioridades.

El primer principio *–la necesidad de conocer–* tiene que ver con la abstracción y la modularización tal como las hemos descrito en el Capítulo 3. Si fuera necesario conocer todos los detalles internos de todas las clases que necesitamos utilizar, nunca terminaríamos de implementar sistemas de gran envergadura.

Concepto:

El **ocultamiento de la información** es un principio que establece que los detalles internos de la implementación de una clase deben estar ocultos a los ojos de otras clases. Garantiza una mejor modularización de una aplicación.

El segundo principio *–que no se permita conocer–* es diferente. También tiene que ver con la modularización, pero en un contexto distinto. Los lenguajes de programación no permiten que las instrucciones de otra clase accedan a la sección privada de una clase. Esto garantiza que ninguna clase dependa de cómo se implemente exactamente otra clase.

Esto es muy importante para el trabajo de mantenimiento. Es una tarea bastante común, para un programador de mantenimiento, modificar o ampliar posteriormente la implementación de una clase, con el fin de realizar mejoras o corregir errores. Idealmente, modificar la implementación de una clase no debería hacer necesario que se modifiquen también otras clases. Esta cuestión se conoce también con el nombre de *acoplamiento*. Si los cambios en una parte de un programa no hacen necesario realizar también cambios en otra parte de un programa, decimos que hay un acoplamiento bajo o un acoplamiento débil. El acoplamiento débil es positivo porque facilita mucho el trabajo de un programador de mantenimiento. En vez de tener que entender y modificar muchas clases, puede que solo necesite entender y modificar una única clase. Por ejemplo, si un programador de sistemas Java hace una mejora en la implementación de la clase `ArrayList`, lo que esperamos es que eso no nos obligue a modificar aquellas partes de nuestro código donde se use esa clase. En principio, debería ser así, porque no hemos hecho ninguna referencia a la implementación de `ArrayList` dentro de nuestro propio código.

Por tanto, para ser más precisos, la regla de que a un usuario no debería permitírsele conocer las interioridades de una clase no se refiere al programador de otra clase, sino a la propia clase. Normalmente, no es ningún problema que un programador conozca los detalles de implementación, pero las clases que ese programador desarrolle no deberían "conocer" (no deberían ser dependientes de) los detalles internos de otra clase. El programador de ambas clases puede ser incluso la misma persona, pero las clases deben estar débilmente acopladas.

Las cuestiones del acoplamiento y del ocultamiento de la información son muy importantes, y hablaremos más de ellas en capítulos posteriores.

Por ahora, es importante comprender que la palabra clave `private` provoca el ocultamiento de la información al no permitir que otras clases accedan a esta parte de la clase. Esto garantiza un acoplamiento débil y hace que una aplicación sea más modular y fácil de mantener.

5.11.2 Métodos privados y campos públicos

La mayoría de los métodos que hemos visto ahora eran públicos. Esto garantiza que otras clases puedan invocarlos. En ocasiones, sin embargo, hemos hecho uso de métodos

privados. Por ejemplo, en la clase `SupportSystem` del sistema *TechSupport*, hemos visto que los métodos `printWelcome` y `printGoodbye` se declaraban como métodos privados.

La razón de disponer de ambas opciones es que los métodos se utilizan, en la práctica, con propósitos distintos. Se emplean para proporcionar operaciones a los usuarios de una clase (métodos públicos), pero también se usan para descomponer una tarea de gran envergadura en otras tareas más pequeñas, con el fin de que la tarea de mayor tamaño sea más manejable. En el segundo caso, las subtareas no están pensadas para ser invocadas directamente desde fuera de la clase, sino que se colocan en métodos separados simplemente para hacer que la implementación de una clase sea más fácil de leer. En este caso, dichos métodos deben ser privados. Los métodos `printWelcome` y `printGoodbye` son ejemplos de este tipo de métodos.

Otra buena razón para tener un método privado es cuando hace falta una tarea (como subtarea) en varios de los métodos de una clase. En lugar de escribir el código múltiples veces, podemos escribirlo una sola vez como un único método privado y luego invocarlo desde varios lugares distintos. Veremos un ejemplo de esto más adelante.

En Java, los campos también pueden declararse como privados o públicos. Hasta ahora, no hemos visto ejemplos de campos públicos y hay una buena razón para ello. Declarar campos como públicos viola el principio de ocultamiento de la información. Hace que una clase que dependa de dicha información sea vulnerable a los fallos de operación, en caso de que la implementación cambie. Aun cuando el lenguaje Java nos permite declarar campos públicos, consideramos que este es un mal estilo de programación, así que no vamos a hacer uso de dicha opción. Algunos otros lenguajes orientados a objetos no permiten en absoluto los campos públicos.

Una razón adicional para mantener privados los campos es que permite a los objetos tener un mayor grado de control sobre su propio estado. Si canalizamos el acceso a un campo privado a través de métodos selectores y mutadores, entonces los objetos tendrán la posibilidad de garantizar que el campo no se configure nunca con un valor que sea incoherente con su estado global. Este nivel de integridad no es posible si hacemos públicos los campos.

En resumen, los campos deben ser siempre privados.

Java tiene dos niveles adicionales de acceso. Uno se declara utilizando la palabra clave `protected` como modificador de acceso; el otro se utiliza si no se declara ningún modificador de acceso en absoluto. Hablaremos de estas opciones en posteriores capítulos.

5.12 Aprendiendo acerca de las clases a partir de sus interfaces

Vamos a analizar brevemente otro proyecto para repasar y practicar los conceptos expuestos en este capítulo. El proyecto se llama *scribble* y puede encontrarlo en la carpeta correspondiente al Capítulo 5 de los proyectos del libro. Esta sección no introduce ningún nuevo concepto, por lo que está compuesta en buena parte por ejercicios, con algún comentario intercalado.

5.12.1 La demo `scribble`

El proyecto *scribble* proporciona tres clases: `DrawDemo`, `Pen` y `Canvas` (Figura 5.4).

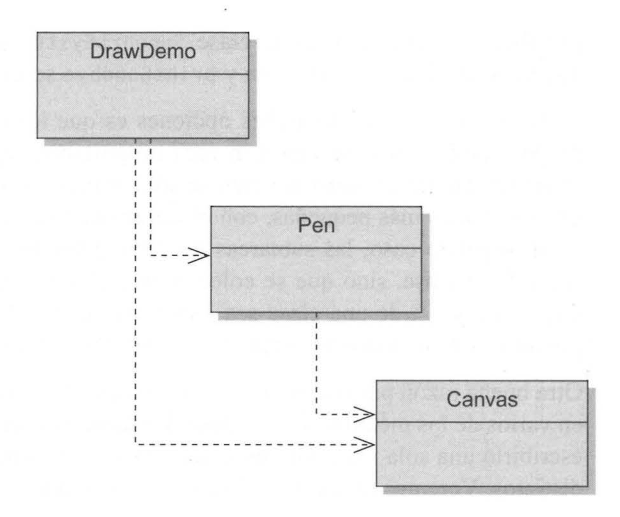


Figura 5.4
El proyecto `scribble`.

La clase `Canvas` proporciona una ventana en la pantalla que se puede utilizar como lienzo para dibujar en él. Dispone de operaciones para dibujar líneas, formas y texto. Puede emplearse un lienzo creando una instancia interactivamente o desde otro objeto. Puede utilizar la clase `Canvas` sin efectuar ninguna modificación. Probablemente, lo mejor sea tratarla como una clase de librería: abra el editor y cambie a la vista de documentación. Aparecerá la interfaz de la clase con la documentación `javadoc`.

La clase `Pen` proporciona un objeto lápiz que puede utilizarse para generar dibujos sobre el lienzo, moviéndolo por la pantalla. El propio lápiz es invisible, pero hará que se dibuje una línea cuando se mueva por el lienzo.

La clase `DrawDemo` proporciona unos pocos ejemplos de cómo utilizar un objeto lápiz para generar un dibujo en la pantalla.

El mejor punto de partida para entender y experimentar con este proyecto es la clase `DrawDemo`.

Ejercicio 5.50 Cree un objeto `DrawDemo` y experimente con sus diversos métodos. Lea el código fuente de `DrawDemo` y describa (por escrito) cómo funciona cada método.

Ejercicio 5.51 Cree un objeto `Pen` interactivamente utilizando su constructor predeterminado (el constructor sin parámetros). Experimente con sus métodos. Mientras lo hace, asegúrese de tener una ventana abierta que le muestre la documentación de la clase `Pen` (bien la ventana del editor en la vista *Documentation* o una ventana del explorador web donde se muestre la documentación del proyecto). Consulte la documentación para asegurarse de lo que hace cada método.

Ejercicio 5.52 Cree interactivamente una instancia de la clase `Canvas` y pruebe algunos de sus métodos. Una vez más, consulte la documentación de la clase mientras experimenta.

Algunos de los métodos de las clases `Pen` y `Canvas` hacen referencia a parámetros de tipo `Color`. Este tipo está definido en la clase `Color` del paquete `java.awt` (por tanto, su nombre completamente cualificado es `java.awt.Color`). La clase `Color` define algunas constantes de color, a las que podemos hacer referencia de la forma siguiente:

```
Color.RED
```

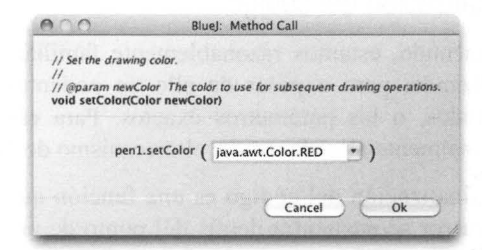
La utilización de estas constantes requiere importar la clase `Color` en la clase que vaya a ser empleada.

Ejercicio 5.53 Encuentre algunos usos de las constantes de color en el código de la clase `DrawDemo`.

Ejercicio 5.54 Escriba cuatro constantes de color adicionales que estén disponibles en la clase `Color`. Consulte la documentación de la clase para ver cuáles pueden ser.

Al invocar de forma interactiva métodos que esperen parámetros de la clase `Color`, tenemos que referirnos a la clase de forma ligeramente distinta. Puesto que el cuadro de diálogo interactivo no tiene ninguna instrucción de importación (y por tanto la clase `Color` no es conocida automáticamente), debemos escribir el nombre de clase completamente cualificado cada vez que hagamos referencia a la clase (Figura 5.5). Esto permite al sistema Java encontrar la clase sin necesidad de utilizar una instrucción de importación.

Figura 5.5



Ahora que sabemos cómo cambiar el color de los lápices y de los lienzos, podemos hacer algunos ejercicios más.

Ejercicio 5.55 Cree un lienzo. Utilizando interactivamente los métodos del lienzo, dibuje un círculo rojo cerca del centro del lienzo. Ahora dibuje un rectángulo amarillo.

Ejercicio 5.56 ¿Cómo se puede borrar el lienzo completo?

Como hemos visto, podemos dibujar directamente en el lienzo o utilizar un objeto lápiz. El lápiz nos proporciona una abstracción que almacena una posición, una rotación y un color actuales, y esto hace que la generación de algunos tipos de dibujos resulte más fácil. Experimentemos con esto un poco más, pero esta vez escribiendo el código de una clase en lugar de utilizando llamadas interactivas.

Ejercicio 5.57 En la clase `DrawDemo`, cree un método nuevo denominado `drawTriangle`. Este método debe crear un lápiz (como en el método `drawSquare`) y luego dibujar un triángulo verde.

Ejercicio 5.58 Escriba un método `drawPentagon` que dibuje un pentágono.

Ejercicio 5.59 Escriba un método `drawPolygon(int n)` que dibuje un polígono regular de `n` lados (por tanto, n=3 dibuja un triángulo, n=4 dibuja un cuadrado, etc.).

Ejercicio 5.60 Escriba un método denominado `spiral` que dibuje una espiral (véase la Figura 5.6).

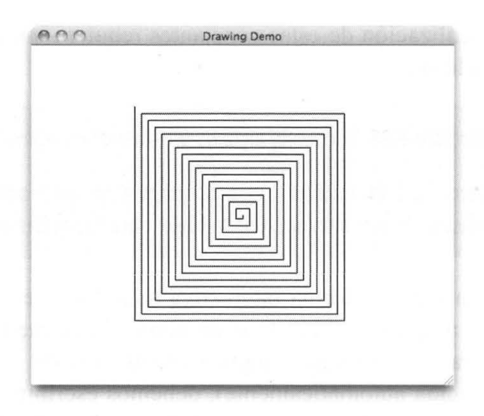


Figura 5.6
Una espiral dibujada sobre el lienzo.

5.12.2 Finalización del código

A menudo, estamos razonablemente familiarizados con una clase de librería que estamos utilizando, pero a pesar de ello no podemos recordar los nombres exactos de todos los métodos, o los parámetros exactos. Para estos casos, los entornos de desarrollo ofrecen normalmente algo de ayuda: el mecanismo de finalización del código.

La finalización del código es una función que está disponible en el editor de BlueJ cuando el cursor se encuentra detrás del punto de una llamada a método. En esta situación, pulsar las teclas *CTRL-espacio* hará que aparezca un cuadro emergente en el que se enumeran todos los métodos existentes en la interfaz del objeto que estamos utilizando en esa llamada (Figura 5.7).

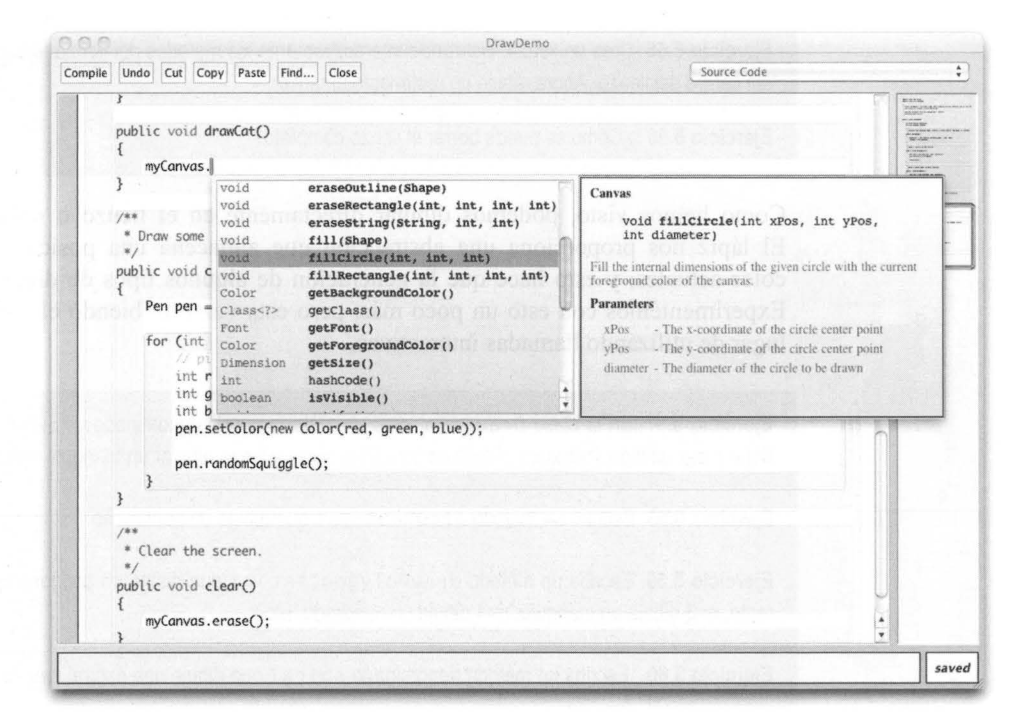


Figura 5.7
Finalización del código.

Cuando se muestra el cuadro emergente de finalización de código, podemos escribir el principio del nombre del método con el fin de filtrar la lista de métodos. Al pulsar la tecla de *retorno de carro* la llamada al método seleccionado se inserta en nuestro código fuente. Esta función de finalización de código también puede utilizarse sin ningún objeto precedente, con el fin de invocar métodos locales.

La utilización del mecanismo de finalización del código no debería tomarse como un sustituto de la lectura de la documentación de una clase, porque no incluye toda la información (como por ejemplo el comentario de introducción a la clase). Pero una vez que estemos razonablemente familiarizados con una clase en general, el mecanismo de finalización del código es de gran ayuda para recordar más fácilmente los detalles de un método e introducir la llamada en nuestro código fuente.

Ejercicio 5.61 Añada un método a su clase `DrawDemo` que genere directamente un dibujo sobre el lienzo (sin utilizar un objeto lápiz). El dibujo puede mostrar cualquier cosa que desee, pero debe incluir al menos algunas formas, diferentes colores y texto. Utilice el mecanismo de finalización del código durante el proceso de escritura de su código fuente.

5.12.3 La demo `bouncing-balls`

Abra el proyecto *bouncing-balls* y averigüe lo que hace. Cree un objeto `BallDemo` y ejecute su método `bounce`.

Ejercicio 5.62 Modifique el método `bounce` de la clase `BallDemo` para dejar que el usuario seleccione cuántas bolas debe haber rebotando.

Para este ejercicio, tendrá que utilizar una colección para almacenar las bolas. De esta forma, el método podrá manejar 1, 3 o 75 bolas (cualquier número que desee). Las bolas deben colocarse inicialmente en un fila a lo largo de la parte superior del lienzo.

¿Qué tipo de colección deberíamos elegir? Hasta el momento, hemos visto un `ArrayList`, un `HashMap` y un `HashSet`. Trate primero de hacer los siguientes ejercicios antes de escribir su implementación.

Ejercicio 5.63 ¿Qué tipo de colección (`ArrayList`, `HashMap` o `HashSet`) es más adecuada para almacenar las bolas para el nuevo método `bounce`? Explíquelo por escrito y justifique su elección.

Ejercicio 5.64 Modifique el método `bounce` para colocar las bolas aleatoriamente en cualquier lugar de la mitad superior de la pantalla.

Ejercicio 5.65 Escriba un nuevo método denominado `boxBounce`. Este método dibuja un rectángulo (la "caja") en la pantalla y una o más bolas dentro la caja. Para las bolas, no utilice `BouncingBall`, cree una nueva clase `BoxBall` que haga que la bola se mueva por el interior de la caja, rebotando en las paredes de la misma de tal forma que la bola siempre quede en el interior. La posición y la velocidad iniciales de la bola deben ser aleatorias. El método `boxBounce` debe tener un parámetro que especifique cuántas bolas hay que meter en la caja.

Ejercicio 5.66 Proporcione colores aleatorios a las bolas de `boxBounce`.

5.13 Variables de clase y constantes

Hasta ahora no hemos examinado la clase `BouncingBall`. Si está realmente interesado en comprender cómo funciona esta animación, puede probar a estudiar también esta clase. Es una clase razonablemente simple; el único método cuya comprensión requiere un cierto esfuerzo es `move`, en el que la bola cambia su posición a la siguiente posición de su trayectoria.

Dejamos principalmente como ejercicio para el lector el estudio de este método, salvo por un detalle que vamos a explicar a continuación. Comenzaremos con un ejercicio.

Ejercicio 5.67 En la clase `BouncingBall`, encontrará una definición de `GRAVITY` (un simple entero que representa la gravedad). Incremente o decremente el valor de la gravedad; compile y ejecute de nuevo la demo de las bolas rebotando. ¿Puede observar algún cambio?

El detalle interesante de esta clase se encuentra en la línea

```
private static final int GRAVITY = 3;
```

Esta es una estructura que todavía no hemos visto. Esta línea de hecho introduce dos nuevas palabras clave que se utilizan conjuntamente: `static` y `final`.

5.13.1 La palabra clave `static`

Concepto:

Las clases pueden tener campos. Estos se conocen con el nombre de **variables de clase** o variables estáticas. De cada variable de clase existirá en todo momento exactamente una copia, independientemente del número de instancias que se creen de dicha clase.

La palabra clave `static` es la sintaxis de Java para definir *variables de clase*. Las variables de clase son campos que se almacenan en la propia clase, no en un objeto. Esto los hace fundamentalmente distintos de las variables de instancia (los campos con los que hemos tratado hasta ahora). Considere el siguiente segmento de código (una parte de la clase `BouncingBall`):

```
public class BouncingBall
{
    // Efecto de la gravedad.
    private static final int GRAVITY = 3;

    private int xPosition;
    private int yPosition;

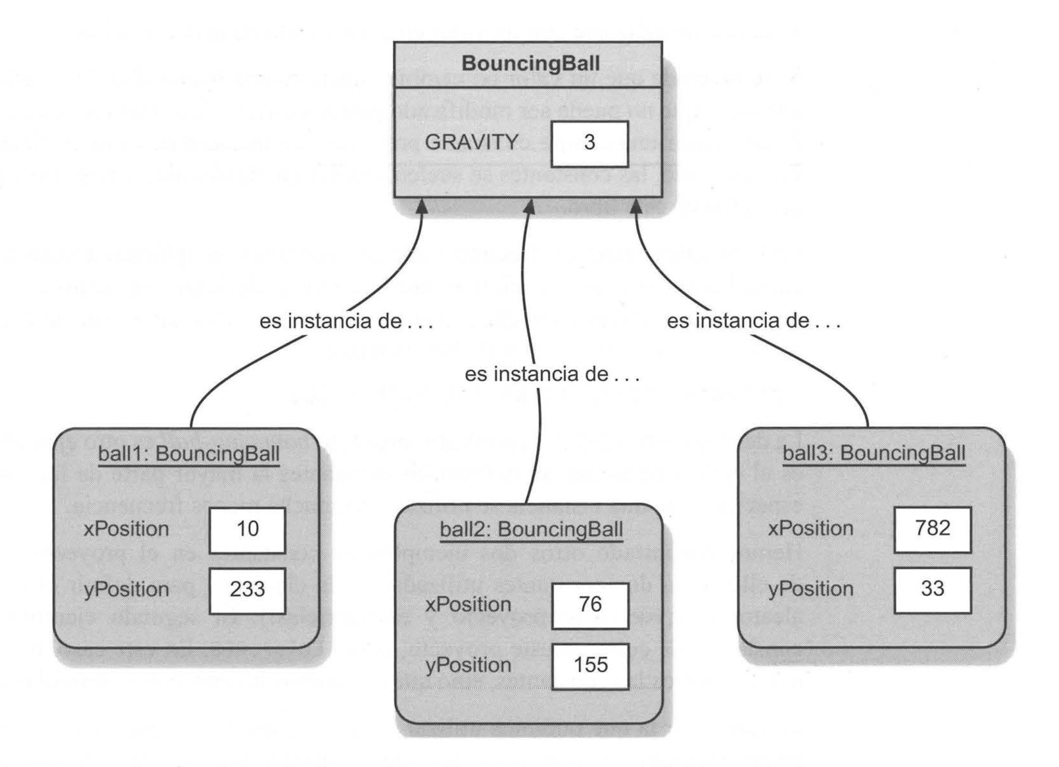
     Se omiten otros campos y métodos.
}
```

Ahora imagine que creamos tres instancias de `BouncingBall`. La situación resultante será la mostrada en la Figura 5.8.

Como podemos ver en el diagrama, las variables de instancia (`xPosition` y `yPosition`) están almacenadas en cada objeto. Como hemos creado tres objetos, dispondremos de tres copias independientes de estas variables.

Por el contrario, la variable de clase `GRAVITY` está almacenada en la propia clase. Como resultado, habrá siempre una única copia de esta variable, independientemente del número de instancias que creemos.

Figura 5.8
Variables de instancia y una variable de clase.



El código fuente de la clase puede acceder (leer y configurar) este tipo de variable exactamente igual que en el caso de una variable de instancia. A la variable de clase se puede acceder desde cualquiera de las instancias de la clase. Como resultado, todos los objetos de esa clase compartirán dicha variable.

Las variables de clase se emplean frecuentemente cuando tenemos un valor que debe ser siempre el mismo para todas las instancias de una clase. En lugar de almacenar una copia del mismo valor en cada objeto, lo que sería un desperdicio de espacio y podría ser difícil de coordinar, se utiliza un único valor compartido por todas las instancias.

Java también soporta los *métodos de clase* (también conocidos como *métodos estáticos*), que son métodos que pertenecen a una clase. Hablaremos de ellos más adelante.

5.13.2 Constantes

Un uso frecuente de la palabra clave `static` es en la definición de *constantes*. Las constantes son similares a las variables, pero no pueden cambiar de valor durante la ejecución de una aplicación. En Java, las constantes se definen mediante la palabra clave `final`. Por ejemplo:

```
private final int SIZE = 10;
```

Aquí, hemos definido una constante denominada `SIZE` con el valor 10. Observe que las declaraciones de constantes tienen un aspecto similar a las declaraciones de campos, con dos diferencias:

- incluyen la palabra clave `final` antes del nombre del tipo y

- deben inicializarse con un valor en el momento de la declaración.

Si se pretende que un valor no cambie nunca, es una buena idea declararlo como `final`. Esto garantiza que no pueda ser modificado posteriormente de manera accidental. Cualquier intento de modificar una campo constante provocará un mensaje de error en tiempo de compilación. Por convenio, las constantes se suelen escribir en mayúsculas y nosotros vamos a seguir dicho convenio en este libro.

En la práctica, suele ser frecuente que las constantes se apliquen a todas las instancias de una clase. En este tipo de situación, lo que hacemos es declarar *constantes de clase*. Las constantes de clase son campos de clase constantes. Se declaran utilizando una combinación de las palabras clave `static` y `final`. Por ejemplo:

```
private static final int SIZE = 10;
```

La definición de `GRAVITY` en nuestro proyecto *bouncing-ball* es otro ejemplo de constante. Este es el estilo con el que se definen las constantes la mayor parte de las veces. Las constantes específicas de una instancia se utilizan con mucha menos frecuencia.

Hemos encontrado otros dos ejemplos de constantes en el proyecto *scribble*. El primero de ellos eran dos constantes utilizadas en la clase `Pen` para definir el tamaño del "garabato aleatorio" (¡vuelva al proyecto y encuéntrelas!). El segundo ejemplo era el uso de las constantes de color en este proyecto, como `Color.RED`. En este caso, no fuimos nosotros los que definimos las constantes, sino que utilizamos las constantes definidas en otra clase.

La razón por la que pudimos utilizar las constantes de la clase `Color` es que están declaradas como públicas. A diferencia de otros campos (acerca de los cuales ya hemos dicho anteriormente que nunca deben declararse como públicos), la declaración de constantes como públicas no suele generar ningún problema y resulta en ocasiones muy útil.

Ejercicio 5.68 Escriba declaraciones de constantes para lo siguiente:

- Una variable pública que se use para medir la tolerancia, con el valor 0.001.
- Una variable privada que se use para indicar una marca de aceptación, con un valor entero igual a 40.
- Una variable de carácter pública que se use para indicar que el comando de ayuda es `'h'`.

Ejercicio 5.69 Eche un vistazo a la clase `LogEntry` en el proyecto *weblog-analyzer* del Capítulo 4. ¿Cómo se han utilizado constantes en esa clase? ¿Cree que ese es un buen uso de las constantes?

Ejercicio 5.70 Suponga que realizáramos una modificación en el proyecto *weblog-analyzer* que implicara que ya no es necesario almacenar los valores del año en la matriz `dataValues` de la clase `LogEntry`. ¿Qué parte de la clase sería necesario modificar si el valor del mes tuviera ahora que almacenarse en el índice `0`, el valor del día en el índice `1` y así sucesivamente? ¿Entiende por qué el uso de constantes nominadas para valores especiales simplifica este tipo de proceso?

5.14 Resumen

Trabajar con librerías de clase y con interfaces de clases es esencial para un programador competente. Hay dos aspectos en esta cuestión: la lectura de las descripciones de las librerías de clases (especialmente de las interfaces de clases) y la escritura de esas descripciones.

Es importante conocer algunas de las clases esenciales de la librería de clases estándar Java y ser capaz de conseguir más información en caso necesario. En este capítulo, hemos presentado algunas de las clases más importantes y hemos explicado como explorar la documentación de la librería.

También es importante ser capaz de documentar cualquier clase que escribamos, con el mismo estilo que se emplea en las clases de librería, con el fin de que otros programadores puedan usar fácilmente la clase sin necesidad de comprender su implementación. Esta documentación debe incluir buenos comentarios para todo proyecto, clase y método. La utilización de `javadoc` con los programas Java le ayudará en esta tarea.

Términos introducidos en el capítulo

interfaz, implementación, mapa, conjunto, javadoc, modificador de acceso, ocultamiento de la información, acoplamiento, variable de clase, static, constante, final

Resumen de conceptos

- **librería Java** La librería de clases estándar de Java contiene muchas clases de gran utilidad. Es importante saber cómo utilizar la librería.
- **documentación de librería** La documentación de la librería de clases Java muestra detalles acerca de todas las clases de la librería. La utilización de esta documentación es esencial para poder hacer un buen uso de las clases de librería.
- **interfaz** La interfaz de una clase describe lo que una clase hace y cómo se la puede utilizar sin mostrar su implementación.
- **implementación** El código fuente completo que define una clase es la implementación de dicha clase.
- **inmutable** Se dice que un objeto es inmutable si su contenido o estado no puede cambiarse después de crearlo. Las cadenas de caracteres son un ejemplo de objeto inmutable.
- **mapa** Un mapa es una colección que almacena parejas clave/valor como entradas. Se pueden buscar valores proporcionando la clave.
- **conjunto** Un conjunto es una colección que almacena cada elemento individual como máximo una vez. No mantiene ningún orden específico.
- **documentación** La documentación de una clase debe ser lo suficientemente detallada como para que otros programadores puedan utilizar la clase sin necesidad de leer su implementación.
- **modificador de acceso** Los modificadores de acceso definen la visibilidad de un campo, constructor, o método. Los elementos públicos son accesibles desde dentro de la misma clase y desde otras clases; los elementos privados solo son accesibles desde dentro de la misma clase.
- **ocultamiento de información** El ocultamiento de la información es un principio que establece que los detalles internos de la implementación de una clase deben estar ocultos a los ojos de otras clases. Garantiza una mejor modularización de una aplicación.
- **variable de clase, variable estática** Las clases pueden tener campos. Estos se conocen con el nombre de variables de clase o variables estáticas. De cada variable de clase existirá en todo momento exactamente una copia, independientemente del número de instancias que se creen de dicha clase.

Ejercicio 5.71 Circula en Internet el rumor de que George Lucas (el creador de la serie de películas *Star Wars*) utiliza una fórmula para crear los nombres de los personajes de sus historias (Jar Jar Binks, ObiWan Kenobi, etc.). La supuesta fórmula es la siguiente:

Tu nombre en Star Wars*:*
1 Toma las tres primeras letras de tu apellido.
2 Añádeles las dos primeras letras de tu nombre.
Tu apellido en Star Wars*:*
1 Toma la dos primeras letras del apellido de tu madre.
2 Añádeles las tres primeras letras del nombre de la ciudad donde naciste.

Y ahora esta es la tarea que le encomendamos: cree un nuevo proyecto en BlueJ denominado *starwars*. En él, cree una clase llamada `NameGenerator`. Esta clase debe tener un método denominado `generateStarWarsName` que genere un nombre *Star Wars*, siguiendo el método descrito anteriormente. Tendrá que localizar información acerca de un método de la clase `String` que genera una subcadena.

Ejercicio 5.72 El siguiente fragmento de código trata de imprimir una cadena en letras mayúsculas:

```
public void printUpper(String s)
{
    s.toUpperCase();
    System.out.println(s);
}
```

Sin embargo, este código no funciona. Averigüe por qué y explique la razón. ¿Cómo se escribiría apropiadamente?

Ejercicio 5.73 Suponga que queremos intercambiar los valores de dos variables enteras, `a` y `b`. Para ello, escribimos un método

```
public void swap(int i1, int i2)
{
    int tmp = i1;
    i1 = i2;
    i2 = tmp;
}
```

Después invocamos este método con nuestras variables `a` y `b`:

```
swap(a, b);
```

¿Se intercambiarán `a` y `b` después de esta llamada? Si lo prueba, observará que no se intercambian. ¿Por qué no funciona esto? Explique su respuesta en detalle.

CAPÍTULO

6 Diseño de clases

Principales conceptos explicados en el capítulo:

- diseño dirigido por responsabilidad
- cohesión
- acoplamiento
- refactorización

Estructuras Java explicadas en este capítulo:

`static` (para métodos), `Math`, tipos enumerados, `switch`

En este capítulo, vamos a echar un vistazo a algunos de los factores que influyen sobre el diseño de una clase. ¿Qué hace que un diseño de una clase sea bueno o malo? Escribir clases adecuadamente puede requerir más esfuerzo a corto plazo que escribirlas de manera incorrecta, pero a largo plazo ese esfuerzo adicional se verá a menudo recompensado. Para ayudarnos a escribir clases de manera adecuada, hay algunos principios que podemos aplicar. En particular, en este capítulo vamos a presentar el punto de vista de que el diseño de clases debe estar dirigido por el concepto de responsabilidad, y que las clases deben encapsular los datos correspondientes.

Este capítulo está, como muchos de los anteriores, estructurado alrededor de un proyecto. Puede estudiarlo limitándose a leerlo y a seguir nuestra línea de razonamiento, o bien lo puede estudiar con mucha mayor profundidad realizando los ejercicios del proyecto en paralelo con la lectura del capítulo.

El trabajo en el proyecto está dividido en tres partes. En la primera parte, hablaremos de los cambios necesarios en el código fuente y desarrollaremos y mostraremos las soluciones completas a los ejercicios. La solución para esta parte está también disponible en uno de los proyectos de acompañamiento del libro. En la segunda parte se sugieren más modificaciones y ampliaciones, y hablaremos de las posibles soluciones a un alto nivel (el nivel de diseño de clases), pero dejaremos como ejercicio para el lector la realización del trabajo de bajo nivel y el remate de la implementación.

En la tercera parte se sugieren aun más mejoras en forma de ejercicios. Allí no proporcionaremos soluciones –los ejercicios sirven para aplicar el material presentado a lo largo del capítulo.

La implementación de todas las partes constituye un buen proyecto de programación para trabajar en él a lo largo de varias semanas. También puede utilizarse muy convenientemente como proyecto de grupo.

6.1 Introducción

Se puede perfectamente implementar una aplicación y hacer que lleve a cabo su tarea con una serie de clases mal diseñadas. El ejecutar una aplicación terminada no suele indicar nada acerca de si está bien estructurada internamente o no.

Los problemas suelen aflorar cuando un programador de mantenimiento quiere hacer más tarde algunas modificaciones en una aplicación existente. Por ejemplo, si un programador trata de corregir un error o quiere añadir una nueva funcionalidad a un programa existente, una tarea que podría resultar muy fácil e inmediata con una serie de clases bien diseñadas puede resultar enormemente complicada y requerir una gran cantidad de trabajo si las clases están mal diseñadas.

En aplicaciones de gran tamaño, estos efectos aparecen en una etapa temprana, durante la implementación original. Si la implementación comienza con una mala estructura, entonces finalizarla puede llegar a ser tremendamente complejo, y el programa completo puede no terminarse, o puede contener errores, o puede requerir mucho más tiempo para su desarrollo de lo que sería necesario. En el mundo real, las empresas suelen mantener, ampliar y comercializar una aplicación a lo largo de muchos años. No es infrecuente que una implementación de un paquete software que podemos adquirir hoy día haya sido iniciada hace más de diez años. En esta situación, una empresa de software no puede permitirse disponer de un código mal estructurado.

Puesto que muchos de los efectos de un mal diseño de clases se hacen especialmente obvios al tratar de adaptar o ampliar una aplicación, lo que vamos a hacer aquí es exactamente eso. En este capítulo, utilizaremos un ejemplo denominado *world-of-zuul*, que es una implementación simple y muy rudimentaria de un juego de aventuras basado en texto. En su estado original, el juego no es realmente muy ambicioso, entre otras cosas, porque está incompleto. Sin embargo, al final del capítulo estaremos en disposición de ejercitar nuestra imaginación y de diseñar e implementar nuestro propio juego, haciéndolo realmente divertido e interesante.

world-of-zuul Nuestro juego *world-of-zuul* está modelado según el juego original *Adventure* que desarrolló a principios de la década de 1970 Will Crowther y que posteriormente fue ampliado por Don Woods. El juego original se conoce también en ocasiones con el nombre de *Colossal Cave Adventure*. Era un juego tremendamente imaginativo y sofisticado para su época, requiriendo que el jugador encontrara el camino a través de un complejo sistema de cuevas, localizando tesoros ocultos, utilizando palabras secretas y otros misterios..., todo ello con el objetivo de conseguir el máximo número de puntos. Puede obtener más información sobre el juego en sitios como `http://jerz.setonhill.edu/if/canon/Adventure.htm` y `http://www.rickadams.org/adventure/`, o buscando en la Web la frase "*Colossal Cave Adventure*".

Mientras trabajemos en la ampliación de la aplicación original, aprovecharemos para explicar algunos aspectos de su diseño de clases existente. Veremos que la implementación con la que vamos a empezar contiene ejemplos de mal diseño, y podremos comprobar cómo afecta esto a nuestras tareas y cómo podemos corregirlo.

En los ejemplos de proyectos del libro, podrá encontrar dos versiones del proyecto *zuul*: *zuul-bad* y *zuul-better*. Ambas implementan exactamente la misma funcionalidad, pero

parte de la estructura es distinta, representando uno de los proyectos un mal diseño y el otro un diseño mejor. El hecho de que podamos implementar la misma funcionalidad de forma adecuada o inadecuada ilustra que los malos diseños no suelen ser consecuencia de la dificultad del problema que queremos resolver. Los malos diseños tienen más que ver con las decisiones que tomamos a la hora de resolver un problema concreto. No podemos utilizar como excusa para hacer un mal diseño el argumento de que no había otra manera de resolver el problema.

Por tanto, vamos a utilizar el proyecto con un diseño inadecuado para poder explorar por qué resulta inadecuado y luego mejorarlo. La otra versión es una implementación de los cambios que vamos a explicar aquí.

Ejercicio 6.1 Abra el proyecto *zuul-bad*. (Este proyecto se denomina *bad* (malo) porque su implementación contiene algunas decisiones de diseño incorrectas, y queremos dejar bien claro que este proyecto no debe utilizarse como ejemplo de buenas prácticas de programación.) Ejecute y explore la aplicación. Los comentarios del proyecto le proporcionarán cierta información acerca de cómo ejecutarlo.

Mientras explora la aplicación, responda a las siguientes cuestiones:

- ¿Qué hace esta aplicación?
- ¿Qué comandos acepta el juego?
- ¿Qué hace cada comando?
- ¿Cuántas salas hay en el escenario?
- Dibuje un mapa de las salas existentes.

Ejercicio 6.2 Después de saber lo que hace la aplicación completa, trate de averiguar lo que hace cada una de las clases individuales. Escriba el propósito de cada clase. Para ello, tendrá que examinar el código fuente. Tenga en cuenta que puede que no comprenda todo el código fuente (y en realidad no necesita comprenderlo). A menudo, basta con leer los comentarios y examinar las cabeceras de los métodos.

6.2 El ejemplo del juego *world-of-zuul*

Con el Ejercicio 6.1, habrá visto que el juego *zuul* no es muy interesante. En realidad, es bastante aburrido en su estado actual. Pero proporciona una buena base para que podamos diseñar e implementar nuestro propio juego, que cabe esperar que sea más interesante.

Comenzaremos analizando las clases existentes en nuestra primera versión y trataremos de averiguar qué es lo que hacen. En la Figura 6.1 se muestra el diagrama de clases. El proyecto muestra cinco clases: `Parser`, `CommandWords`, `Command`, `Room` y `Game`. Una investigación del código fuente muestra que, afortunadamente, estas clases están muy bien documentadas, lo que nos permite hacernos una idea inicial de lo que hacen simplemente leyendo el comentario incluido al principio de cada clase (esto también sirve para ilustrar que un mal diseño implica algo más profundo que simplemente el aspecto de una clase o lo buena que sea su documentación). Nuestra comprensión del juego se verá ayudado echando un vistazo al código fuente para ver qué métodos tiene cada clase y qué es lo que algunos de los métodos parecen hacer. He aquí un resumen del propósito de cada clase:

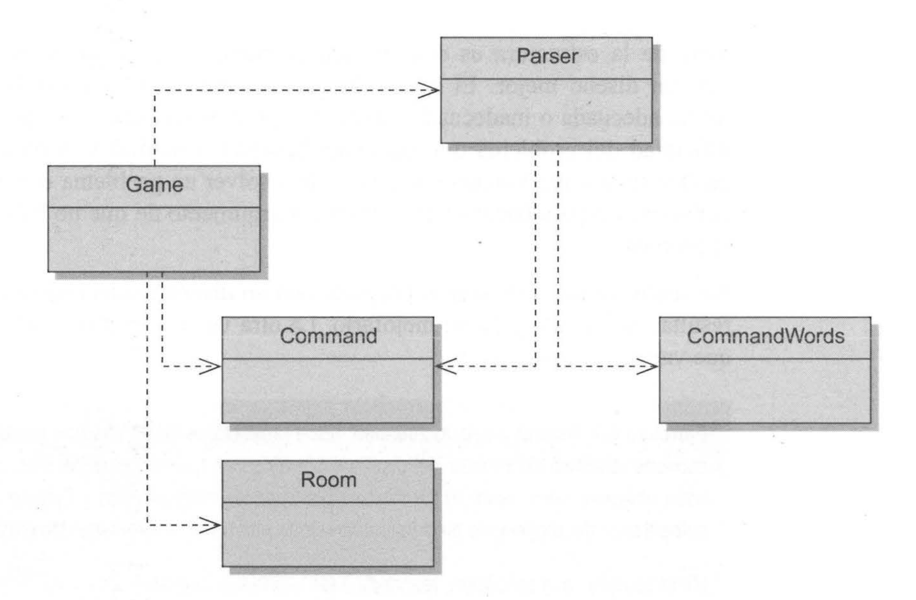


Figura 6.1
Diagrama de clases de `Zuul`.

- *CommandWords* La clase `CommandWords` define todos los comandos válidos del juego. Lo hace manteniendo una matriz de objetos `String` que representan las palabras utilizadas como comandos.
- *Parser* Es el analizador sintáctico, que lee líneas de entrada del terminal y trata de interpretarlas como comandos. Crea objetos de la clase `Command` que representan el comando que se ha introducido.
- *Command* Un objeto `Command` representa un comando introducido por el usuario. Dispone de métodos que hacen que sea fácil comprobar si es un comando válido y extraer la primera y segunda palabras del mismo en forma de cadenas de caracteres independientes.
- *Room* Un objeto `Room` representa una de las ubicaciones o salas del juego. Las salas pueden tener salidas que conducen a otras salas.
- *Game* La clase `Game` es la clase principal del juego. Configura el juego y luego entra en un bucle para leer y ejecutar los comandos. También contiene el código que implementa cada comando de usuario.

Ejercicio 6.3 Diseñe su propio escenario de juego. Hágalo sin utilizar la computadora. No piense en la implementación ni en las clases, ni siquiera en la programación en general. Piense únicamente en inventar un juego interesante. Puede hacer esto con un grupo de personas.

El juego puede ser cualquier cosa que tenga como estructura básica un jugador que se desplaza a través de diferentes ubicaciones. He aquí algunos ejemplos:

- Eres una célula blanca que viaja por el cuerpo buscando virus a los que atacar...
- Estás perdido en un centro comercial y tienes que encontrar la salida...
- Eres un topo en su madriguera y no puedes recordar dónde has almacenado las reservas de comida para el invierno...
- Eres un aventurero que está explorando una mazmorra llena de monstruos y otros personajes...

- Formas parte del equipo de desactivación de explosivos y debes encontrar y desactivar una bomba antes de que estalle...

Asegúrese de que su juego tenga un objetivo (de modo que tenga un final y el jugador pueda "ganar"). Trate de pensar en diversas cosas que hagan que el juego sea interesante (puertas trampa, objetos mágicos, personajes que ayudan al jugador solo si se les alimenta, límites temporales... lo que quiera). Deje que su imaginación vuele libremente.

En esta etapa no se preocupe por cómo implementar esas características.

6.3 Introducción al acoplamiento y a la cohesión

Concepto:

El término **acoplamiento** describe la interconexión de las clases. Lo que buscamos en un sistema es un acoplamiento débil; es decir, un sistema en el que cada clase sea fundamentalmente independiente y se comunique con las otras clases a través de una interfaz compacta y bien definida.

Concepto:

El término **cohesión** describe lo bien que una unidad de código se corresponde con una tarea lógica o con una entidad. En un sistema muy cohesionado, cada unidad de código (método, clase o módulo) es responsable de una tarea o entidad bien definidas. Un buen diseño de clases exhibe un alto grado de cohesión.

Si queremos justificar nuestra afirmación de que unos diseños son mejores que otros, entonces tenemos que definir algunos términos que nos permitan analizar los aspectos que consideramos importantes en el diseño de clases. Hay dos términos que son fundamentales a la hora de hablar de la calidad de un diseño de clase: el *acoplamiento* y la *cohesión*.

El término *acoplamiento* hace referencia al grado de interconexión de las clases. Ya hemos visto en capítulos anteriores que lo que buscamos es diseñar nuestra aplicación como un conjunto de clases en cooperación, que se comunican a través de interfaces bien definidas. El grado de acoplamiento indica lo estrechamente conectadas que están clases. Lo que buscamos es un grado bajo de acoplamiento, o un *acoplamiento débil*.

El grado de acoplamiento determina lo difícil que es realizar cambios en una aplicación. En una estructura de clases estrechamente acoplada, un cambio en un clase puede hacer que sea necesario hacer cambios también en otras clases. Esto es precisamente lo que tratamos de evitar, porque el efecto de realizar un pequeño cambio puede propagarse rápidamente en cascada a través de toda la aplicación. Además, localizar todos los lugares en los que es necesario hacer cambios y llevar a la práctica esos cambios puede resultar difícil y requerir mucho tiempo.

En un sistema débilmente acoplado, por el contrario, podemos cambiar una clase sin efectuar ningún cambio en las restantes clases, y la aplicación seguirá funcionando correctamente. A lo largo del capítulo veremos ejemplos concretos de acoplamiento fuerte y débil.

El término *cohesión* se relaciona con el número y diversidad de las tareas de las que es responsable cada unidad de una aplicación. La cohesión es relevante tanto para unidades formadas por una sola clase, como para métodos individuales.[1]

Idealmente, cada unidad de código debe ser responsable de una tarea coherente (es decir, una tarea que pueda ser vista como una unidad lógica). Cada método debería implementar una operación lógica, y cada clase debería representar un tipo de entidad. La principal razón que subyace al principio de la cohesión es la reutilización: si un método o clase es responsable de

---

[1] En ocasiones, empleamos el término *módulo* (o *paquete* en Java) para hacer referencia a una unidad multiclase. La *cohesión* también es relevante en este nivel.

una única cosa bien definida, entonces es mucho más probable que pueda utilizarse de nuevo en un contexto distinto. Una ventaja complementaria de adherirse a este principio es que, cuando haga falta realizar modificaciones en algún aspecto de la aplicación, es probable que encontremos todas las piezas relevantes dentro de una misma unidad.

Más adelante explicaremos a través de una serie de ejemplos cómo influye la cohesión sobre la calidad del diseño de una clase.

Ejercicio 6.4 Dibuje (en papel) un mapa para el juego que haya inventado en el Ejercicio 6.3. Abra el proyecto *zuul-bad* y guárdelo con otro nombre (por ejemplo, *zuul*). Este es el proyecto que utilizará para realizar mejoras y modificaciones a lo largo del capítulo. Puede olvidarse del sufijo *bad*, porque su proyecto (al menos en teoría) ya no va a ser un mal proyecto en absoluto.

En primer lugar, modifique el método `createRooms` de la clase `Game` para crear las salas y salidas que haya inventado para su juego. ¡Pruebe los cambios!

6.4 Duplicación de código

Concepto:

La **duplicación de código** (tener el mismos segmento de código en una aplicación más de una vez) es un signo de mal diseño. Se debe intentar evitar.

La duplicación de código es un indicador de un mal diseño. La clase `Game` mostrada en el Código 6.1 contiene un caso de duplicación de código. El problema con la duplicación de código es que cualquier modificación en una versión debe ser realizado también en la otra, si queremos evitar las incoherencias. Esto incrementa la cantidad de trabajo que el programador de mantenimiento tiene que realizar e introduce el peligro de que aparezcan errores. Puede suceder muy fácilmente que un programador de mantenimiento encuentre una copia del código y, habiéndola modificado, suponga que ha terminado su trabajo. No hay nada que indique que existe una segunda copia del código, y esa segunda copia podría, incorrectamente, permanecer sin modificaciones.

Código 6.1
Secciones seleccionadas de la clase `Game` (mal diseñada).

```
public class Game
{
    // ... Se omite parte del código...

    private void createRooms()
    {
        Room outside, theater, pub, lab, office;

        // Crear las salas
        outside = new Room(
                "outside the main entrance of the university");
        theater = new Room("in a lecture theater");
        pub = new Room("in the campus pub");
        lab = new Room("in a computing lab");
        office = new Room("in the computing admin office");
```

Código 6.1
(*continuación*)
Secciones seleccionadas de la clase `Game` (mal diseñada).

```
        // Inicializar las salidas de las salas
        outside.setExits(null, theater, lab, pub);
        theatre.setExits(null, null, null, outside);
        pub.setExits(null, outside, null, null);
        lab.setExits(outside, office, null, null);
        office.setExits(null, null, null, lab);

        currentRoom = outside; // comenzar el juego fuera
    }

    // ... Se omite parte del código ...
    /**
     * Imprimir el mensaje de bienvenida para el jugador.
     */
    private void printWelcome()
    {
        System.out.println();
        System.out.println("Welcome to the World of Zuul!");
        System.out.println(
            "Zuul is a new, incredibly boring adventure game.");
        System.out.println("Type 'help' if you need help.");
        System.out.println();
        System.out.println("You are " +
                           currentRoom.getDescription());
        System.out.print("Exits: ");
        if(currentRoom.northExit != null) {
            System.out.print("north ");
        }
        if(currentRoom.eastExit != null) {
            System.out.print("east ");
        }
        if(currentRoom.southExit != null) {
            System.out.print("south ");
        }
        if(currentRoom.westExit != null) {
            System.out.print("west ");
        }
        System.out.println();
    }

    // ... Se omite parte del código...

    /**
     * Tratar de ir en una dirección. Si hay una salida, entrar
     * en la nueva sala; en caso contrario, imprimir un
     * mensaje de error.
     */
```

Código 6.1
(*continuación*)
Secciones seleccionadas de la clase `Game` (mal diseñada).

```
    private void goRoom(Command command)
    {
        if(!command.hasSecondWord()) {
            // Si no hay una segunda palabra,
            // no sabemos a dónde ir
            System.out.println("Go where?");
            return;
        }
        String direction = command.getSecondWord();

        // Tratar de salir de la sala actual.
        Room nextRoom = null;
        if(direction.equals("north")) {
            nextRoom = currentRoom.northExit;
        }
        if(direction.equals("east")) {
            nextRoom = currentRoom.eastExit;
        }
        if(direction.equals("south")) {
            nextRoom = currentRoom.southExit;
        }
        if(direction.equals("west")) {
            nextRoom = currentRoom.westExit;
        }
        if(nextRoom == null) {
            System.out.println("There is no door!");
        }
        else {
            currentRoom = nextRoom;
            System.out.println("You are " +
                                 currentRoom.getDescription());
            System.out.print("Exits: ");
            if(currentRoom.northExit != null) {
                System.out.print("north ");
            }
            if(currentRoom.eastExit != null) {
                System.out.print("east ");
            }
            if(currentRoom.southExit != null) {
                System.out.print("south ");
            }
            if(currentRoom.westExit != null) {
                System.out.print("west ");
            }
            System.out.println();
        }
    }
    // ... Se omite parte del código ...
}
```

Tanto el método `printWelcome` como `goRoom` contienen las siguientes líneas de código:

```
System.out.println("You are " + currentRoom.getDescription());
System.out.print("Exits: ");
if(currentRoom.northExit != null) {
    System.out.print("north ");
}
if(currentRoom.eastExit != null) {
    System.out.print("east ");
}
if(currentRoom.southExit != null) {
    System.out.print("south ");
}
if(currentRoom.westExit != null) {
    System.out.print("west ");
}
System.out.println();
```

Normalmente, la duplicación de código es un síntoma de una mala cohesión. El problema aquí tiene su raíz en el hecho de que ambos métodos hacen dos cosas: `printWelcome` imprime el mensaje de bienvenida y la información acerca de la ubicación actual, mientras que `goRoom` cambia la ubicación actual y luego imprime información acerca de la (nueva) ubicación actual.

Ambos métodos imprimen información sobre la ubicación actual, pero ninguno de ellos puede invocar al otro, porque cada uno de ellos hace también otras cosas. Esto es un ejemplo de un mal diseño.

Un diseño mejor utilizaría un método separado más cohesionado, cuya única tarea sea la de imprimir la información acerca de la ubicación actual (Código 6.2). Tanto el método `printWelcome` como el método `goRoom` pueden entonces invocar este método cada vez que necesiten imprimir dicha información. De esta forma, evitamos escribir el código dos veces, y cuando necesitemos modificarlo, solo tendremos que cambiarlo una vez.

Código 6.2
`printLocationInfo` como método independiente.

```
private void printLocationInfo()
{
    System.out.println("You are " + currentRoom.getDescription());
    System.out.print("Exits: ");
    if(currentRoom.northExit != null) {
        System.out.print("north ");
    }
    if(currentRoom.eastExit != null) {
        System.out.print("east ");
    }
    if(currentRoom.southExit != null) {
        System.out.print("south ");
    }
    if(currentRoom.westExit != null) {
        System.out.print("west ");
    }
    System.out.println();
}
```

Ejercicio 6.5 Implemente y utilice un método independiente `printLocationInfo` en su proyecto como se ha explicado en esta sección. Pruebe las modificaciones realizadas.

6.5 Cómo hacer ampliaciones

El proyecto *zuul-bad* funciona, ciertamente. Podemos ejecutarlo y hace correctamente todo lo que se pretende que haga. Sin embargo, en algunos aspectos está muy mal diseñado. Una alternativa bien diseñada funcionaría de la misma manera; no podríamos observar ninguna diferencia simplemente ejecutando el programa.

Sin embargo, en cuanto tratemos de hacer modificaciones en el proyecto, experimentaremos diferencias significativas en la cantidad de trabajo requerido para modificar un código mal diseñado si lo comparamos con los cambios realizados en una aplicación con un buen diseño. Vamos a investigar este aspecto realizando algunos cambios en el proyecto. Mientras lo hacemos, comentaremos ejemplos de un mal diseño cuando los veamos en el código fuente existente, y mejoraremos el diseño de las clases antes de implementar nuestras ampliaciones.

6.5.1 La tarea

La primera tarea que intentaremos será añadir una nueva dirección de movimiento. Actualmente, el jugador puede moverse en cuatro direcciones: *north*, *east*, *south* y *west*, que se corresponden con los cuatro puntos cardinales, norte, este, sur y oeste. Lo que queremos es permitir edificios de varios niveles (como sótanos o mazmorras, o lo que quiera que deseemos añadir a nuestro juego) y añadir *up* y *down* como posibles direcciones, para movernos hacia arriba y hacia abajo. El jugador podrá entonces escribir `"go down"` para ir, por ejemplo, a un sótano.

6.5.2 Localización del código fuente relevante

La inspección de las clases proporcionadas nos muestra que en este cambio hay involucradas al menos dos clases: `Room` y `Game`.

`Room` es la clase que almacena (entre otras cosas) las salidas de cada sala y, como vimos en el Código 6.1, en la clase `Game` se utiliza la información de salida de la sala actual para imprimir la información acerca de las salidas y para desplazarse de una sala a otra.

La clase `Room` es bastante corta. Su código fuente se muestra en el Código 6.3. Leyendo el código, podemos ver que las salidas se mencionan en dos lugares distintos: se enumeran como campos al comienzo de la clase y también se hacen asignaciones a ellas en el método `setExits`. Para añadir dos nuevas direcciones, tendríamos que añadir dos nuevas salidas (`upExit` y `downExit`) en estos dos lugares.

Requiere algo más de trabajo localizar todos los lugares relevantes dentro la clase `Game`. El código fuente es algo más largo (no lo mostramos completo aquí) y encontrar todos los lugares relevantes requiere algo de paciencia y de cuidado.

Leyendo el código mostrado en el Código 6.1, podemos ver que la clase `Game` utiliza de forma intensiva la información de salida de cada sala. El objeto `Game` almacena una referencia a la

Código 6.3
Código fuente de la clase Room (mal diseñada).

```
public class Room
{
    public String description;
    public Room northExit;
    public Room southExit;
    public Room eastExit;
    public Room westExit;

    /**
     * Crear una sala con una descripción. Inicialmente,
     * no tiene ninguna salida. La descripción es algo como
     * "una cocina" o "un patio abierto".
     */
    public Room(String description)
    {
        this.description = description;
    }

    /**
     * Definir las salidas de esta sala. Cada dirección conduce
     * a otra sala o tiene el valor null (no hay salida ahí).
     */
    public void setExits(Room north, Room east, Room south,
                         Room west)
    {
        if(north != null) {
            northExit = north;
        }
        if(east != null) {
            eastExit = east;
        }
        if(south != null) {
            southExit = south;
        }
        if(west != null) {
            westExit = west;
        }
    }

    /**
     * Devuelve la descripción de la sala (la que se haya
     * definido en el constructor).
     */
    public String getDescription()
    {
        return description;
    }
}
```

sala actual en la variable `currentRoom` y accede frecuentemente a la información de salida de dicha sala.

- En el método `createRoom`, se definen las salidas.
- En el método `printWelcome`, se imprimen las salidas de la sala actual, para que el jugador sepa a dónde ir cuando comienza el juego.
- En el método `goRoom`, se utilizan las salidas para encontrar la siguiente sala. Después se vuelven a utilizar para imprimir las salidas de la nueva sala en la que acabamos de entrar.

Si ahora queremos añadir dos nuevas direcciones de salida, tendremos que agregar las opciones *up* y *down* en todos estos lugares. No obstante, lea la siguiente sección antes de hacer esto.

6.6 Acoplamiento

El hecho de que haya tantos lugares en los que se enumeran todas las salidas es sintomático de un diseño de clases muy pobre. Al declarar las variables de salida en la clase `Room`, necesitamos añadir una variable por cada salida; en el método `setExits`, hay una instrucción if por cada salida; en el método `goRoom`, hay una instrucción if por cada salida; en el método `printLocationInfo`, hay una instrucción if por cada salida; y así sucesivamente. Esta decisión de diseño lo que hace es darnos ahora más trabajo: al añadir nuevas salidas, tenemos que localizar todos esos lugares y añadir dos nuevos casos. ¡Imagine el efecto si decidiéramos utilizar direcciones como el noroeste, el sudeste, etc.!

Para mejorar la situación, decidimos utilizar un `HashMap` para almacenar las salidas, en lugar de emplear variables separadas. De este modo, deberíamos poder escribir código que pueda manejar cualquier número de salidas que deseemos y que no requiera tantas modificaciones. El `HashMap` contendrá una asignación para cada dirección nominada (por ejemplo, `"north"`) que vinculará esa dirección con la sala que haya en esa dirección (un objeto `Room`). Por tanto, cada entrada utilizará un `String` como clave y un objeto `Room` como valor.

Esto representa un cambio en la forma en que cada sala almacena internamente la información acerca de las salas vecinas. En teoría, este es un cambio que solo debería afectar a la *implementación* de la clase `Room` (a *cómo* está almacenada la información acerca de las salidas) y no a la *interfaz* (es decir, a *qué* almacena la sala).

Idealmente, cuando solo cambia la implementación de una clase, otras clases no deberían verse afectadas. Al menos, eso es lo que sucede cuando tenemos un acoplamiento *débil*.

En nuestro ejemplo, esto no funciona. Si eliminamos las variables de salida en la clase `Room` y las sustituimos por un `HashMap`, la clase `Game` ya no podrá compilarse, ya que hace numerosas referencias a las variables de salida de las salas, y esas referencias darán lugar a errores.

Podemos ver que lo que aquí tenemos es un ejemplo de acoplamiento *fuerte*. Para poder corregir esto, vamos a desacoplar estas clases antes de introducir el `HashMap`.

6.6.1 Utilización de la encapsulación para reducir el acoplamiento

Uno de los principales problemas en este ejemplo es el uso de campos públicos. Los campos utilizados para las salidas en la clase `Room` se han declarado todos ellos como

Concepto:

Una adecuada **encapsulación** de las clases reduce el acoplamiento y conduce, por tanto, a un mejor diseño.

`public`. Claramente, el programador de esta clase no ha seguido las directrices que hemos establecido anteriormente en el libro ("¡Nunca hacer públicos los campos!"). Ahora vamos a ver cuál es el resultado. La clase `Game` en este ejemplo puede acceder directamente a esos campos (y hace un uso intensivo de ese hecho). Al hacer públicos los campos, la clase `Room` ha expuesto en su interfaz no solo el hecho de que dispone de salidas, sino también cómo está almacenada exactamente la información acerca de esas salidas. Esto viola uno de los principios fundamentales de un buen diseño de clases: la *encapsulación*.

La directriz referida a la encapsulación (ocultar la información de la implementación a ojos de otras clases) sugiere que solo debe hacerse visible para el exterior la información acerca de *lo que hace una clase*, no la información acerca de *cómo lo hace*. Esto tiene una gran ventaja: si ninguna otra clase sabe cómo está almacenada nuestra información, entonces podemos cambiar fácilmente el modo en que está almacenada sin por ello hacer que otras clases dejen de funcionar.

Podemos obligar a esta separación entre *lo que se hace* y *cómo se hace* definiendo los campos como privados y utilizando un método selector para acceder a ellos. La primera etapa de la clase `Room` modificada se muestra en el Código 6.4.

Código 6.4
Utilización de un método selector para reducir el acoplamiento.

```
public class Room
{
    private String description;
    private Room northExit;
    private Room southExit;
    private Room eastExit;
    private Room westExit;

    // métodos existentes no modificados

    public Room getExit(String direction)
    {
        if(direction.equals("north")) {
            return northExit;
        }
        if(direction.equals("east")) {
            return eastExit;
        }
        if(direction.equals("south")) {
            return southExit;
        }
        if(direction.equals("west")) {
            return westExit;
        }
        return null;
    }
}
```

Habiendo hecho este cambio en la clase `Room`, necesitamos modificar también la clase `Game`. En todos los lugares en los que se utilizaba una variable de salida, vamos a usar ahora el método selector. Por ejemplo, en lugar de escribir

```
nextRoom = currentRoom.eastExit;
```

ahora escribiremos

```
nextRoom = currentRoom.getExit("east");
```

Esto hace también que la codificación de una sección de la clase `Game` sea mucho más fácil. En el método `goRoom`, la modificación sugerida aquí dará como resultado el siguiente segmento de código:

```
Room nextRoom = null;
if(direction.equals("north")) {
    nextRoom = currentRoom.getExit("north");
}
if(direction.equals("east")) {
    nextRoom = currentRoom.getExit("east");
}
if(direction.equals("south")) {
    nextRoom = currentRoom.getExit("south");
}
if(direction.equals("west")) {
    nextRoom = currentRoom.getExit("west");
}
```

Y podemos ver que ahora todo este segmento de código se puede sustituir por:

```
Room nextRoom = currentRoom.getExit(direction);
```

Ejercicio 6.6 Realice los cambios que hemos descrito para las clases `Room` y `Game`.

Ejercicio 6.7 Haga un cambio similar en el método `printLocationInfo` de `Game`, de modo que los detalles de las salidas sean preparados ahora por la instancia de `Room` en lugar de por la instancia de `Game`. Defina un método en `Room` con la siguiente signatura:

```
/**
 * Devuelve una descripción de las salidas de la sala,
 * por ejemplo, "Exits: north west".
 * @return Una descripción de las salidas disponibles.
 */
public String getExitString()-
```

Hasta ahora, no hemos cambiado la representación de las salidas en la clase `Room`, tan solo hemos limpiado un poco la interfaz. El *cambio* en la clase `Game` es mínimo –en lugar de acceder a un campo público, usamos una llamada a método– pero la *mejora* es enorme. Ahora podemos modificar la forma en la que se almacenan las salidas en la sala, sin necesidad de preocuparnos por la posibilidad de que deje de funcionar alguna cosa en la clase `Game`. La representación interna en `Room` se ha desacoplado completamente de la interfaz. Ahora que el diseño es la forma en que debería haber sido desde el principio, sustituir los campos de salida independientes por un `HashMap` es sencillo. El código modificado se muestra en el Código 6.5.

Código 6.5
Código fuente de la clase `Room`.

```
import java.util.HashMap;

// Se omite el comentario de la clase

public class Room
{
    private String description;
    private HashMap<String, Room> exits;

    /**
     * Crear una sala con una descripción. Inicialmente
     * no tiene salidas. La descripción es algo como
     * "una cocina" o "un patio abierto".
     */
    public Room(String description)
    {
        this.description = description;
        exits = new HashMap<String, Room>();
    }

    /**
     * Definir las salidas de esta sala. Cada dirección conduce
     * a otra sala o tiene el valor null (no hay salida ahí).
     */
    public void setExits(Room north, Room east, Room south,
                         Room west)
    {
        if(north != null)
            exits.put("north", north);
        if(east != null)
            exits.put("east", east);
        if(south != null)
            exits.put("south", south);
        if(west != null)
            exits.put("west", west);
    }

    /**
     * Devuelve la sala a la que se llega si nos movemos desde
     * esta sala en la dirección indicada. Si no hay ninguna sala
     * en esa dirección se devuelve null.
     */
    public Room getExit(String direction)
    {
        return exits.get(direction);
    }

    /**
```

Código 6.5
(continuación)
Código fuente de la clase Room.

```
     * Devuelve la descripción de la sala (la que se haya
     * definido en el constructor).
     */
    public String getDescription()
    {
        return description;
    }
}
```

Merece la pena resaltar de nuevo que ahora podemos realizar esta modificación sin ni siquiera molestarnos en comprobar si hay algo que haya dejado de funcionar en alguna otra parte. Puesto que solo hemos cambiado aspectos privados de la clase Room, los cuales, por definición, no pueden utilizarse en otras clases, esta modificación no tendrá ninguna influencia sobre otras clases. La interfaz permanece como estaba.

Un consecuencia indirecta de esta modificación es que ahora nuestra clase Room es incluso más corta. En lugar de enumerar cuatro variables independientes, solo tenemos una. Además, el método getExit se simplifica considerablemente.

Recuerde que el objetivo original que motivó esta serie de cambios era que fuese más fácil añadir las dos salidas posibles en las direcciones *up* y *down*. Esto ahora resulta bastante más fácil. Como ahora estamos empleando un HashMap para almacenar las salidas, se pueden almacenar esas dos direcciones adicionales y la aplicación funcionará sin necesidad de efectuar ningún cambio. También podemos obtener la información de las salidas por medio del método getExit sin ningún problema.

El único lugar que sigue estando codificado en el código fuente el conocimiento acerca de las cuatro salidas existentes (*north*, *east*, *south*, *west*) es en el método setExits. Esta es la última parte que necesitamos cambiar. Por el momento, la signatura del método es

```
public void setExits(Room north, Room east, Room south, Room west)
```

Este método forma parte de la interfaz de la clase Room, por lo que cualquier cambio que realicemos en ella afectará inevitablemente a otras clases debido al acoplamiento. Merece la pena resaltar que nunca podemos desacoplar completamente entre sí las clases de una aplicación; si lo hiciéramos, los elementos de las distintas clases no serían capaces de interactuar entre sí. En lugar de ello, lo que intentamos es mantener el grado de acoplamiento lo más bajo posible. Si tenemos que hacer un cambio en setExits de todos modos, para dar cabida a las direcciones de movimiento adicionales, entonces la solución más adecuada consistirá en sustituirlo completamente por este otro método:

```
/**
 * Definir una salida para esta sala.
 * @param direction La dirección de la salida.
 * @param neighbor La sala que se encuentra en la dirección indicada.
 */
public void setExit(String direction, Room neighbor)
{
    exits.put(direction, neighbor);
}
```

Ahora, las salidas de esta sala pueden configurarse de una en una, pudiéndose emplear cualquier dirección para cada salida. En la clase `Game`, el cambio que resulta de la modificación de la interfaz de `Room` es el siguiente. En lugar de escribir

```
lab.setExits(outside, office, null, null);
```

ahora escribiremos

```
lab.setExit("north", outside);
lab.setExit("east", office);
```

Con esto, hemos eliminado completamente de `Room` la restricción de que solo pueda almacenar cuatro salidas. La clase `Room` está ahora lista para almacenar salidas *up* y *down*, así como cualquier otra dirección en la que podamos pensar (*northwest*, *southeast*, etc.).

Ejercicio 6.8 Implemente los cambios descritos en esta sección en su propio proyecto *zuul*.

6.7 Diseño dirigido por responsabilidad

Concepto:

El **diseño dirigido por responsabilidad** es el proceso de diseñar clases asignando unas reponsabilidades bien definidas a cada clase. Este proceso puede emplearse para determinar qué clase debería implementar cada parte de una función de la aplicación.

Hemos visto en la sección anterior que el hacer uso de una encapsulación adecuada reduce el acoplamiento y puede disminuir significativamente la cantidad de trabajo necesario para llevar a cabo cambios en una aplicación. Sin embargo, la encapsulación no es el único factor que influye en el grado de acoplamiento. Otro aspecto es el conocido con el nombre de *diseño dirigido por responsabilidad.*

El diseño dirigido por responsabilidad expresa la idea de que cada clase debe ser responsable de gestionar sus propios datos. A menudo, cuando necesitamos añadir alguna nueva funcionalidad a una aplicación, tenemos que preguntarnos a nosotros mismos en qué clase deberíamos añadir un método para implementar esa nueva función. ¿Qué clase debería ser responsable de la tarea? La respuesta es que la clase responsable de almacenar unos determinados datos debería ser también responsable de manipularlos.

Lo bien que se utilice el diseño dirigido por responsabilidad influye en el grado de acoplamiento y, por tanto, influye de nuevo en la facilidad con la que se puede modificar o ampliar una aplicación. Como de costumbre, analizaremos esto en mayor detalle con nuestro ejemplo.

6.7.1 Responsabilidades y acoplamiento

Los cambios en la clase `Room` que hemos visto en la Sección 6.6.1 hacen que sea muy sencillo añadir las nuevas direcciones para los movimientos hacia arriba y hacia abajo en la clase `Game`. Vamos a investigar esto con un ejemplo. Suponga que queremos añadir una nueva sala (el sótano, en inglés *cellar*) debajo de la oficina. Todo lo que tenemos que hacer para conseguirlo es efectuar algunos pequeños cambios en el método `createRooms` de `Game`, con el fin de crear la sala y hacer dos llamadas para definir las salidas:

```
private void createRooms()
{
    Room outside, theater, pub, lab, office, cellar;
    ...
```

```
        cellar = new Room("in the cellar");
        ...
        office.setExit("down", cellar);
        cellar.setExit("up", office);
    }
```

Gracias a la nueva interfaz de la clase Room, esto funcionará sin ningún problema. El cambio es ahora muy fácil y confirma que el diseño está mejorando.

Podemos corroborar esta idea todavía más comparando la versión original del método printLocationInfo mostrado en el Código 6.2 con el método getExitString mostrado en el Código 6.6, que representa una solución al Ejercicio 6.7.

Código 6.6
El método getExitString de Room.

```
/**
 * Devuelve una descripción de las salidas de la sala,
 * por ejemplo, "Exits: north west".
 * @return Una descripción de las salidas disponibles.
 */
public String getExitString()
{
    String exitString = "Exits: ";
    if(northExit != null)
        exitString += "north ";
    if(eastExit != null)
        exitString += "east ";
    if(southExit != null)
        exitString += "south ";
    if(westExit != null)
        exitString += "west ";
    return exitString;
}
```

Puesto que la información acerca de sus salidas ahora está almacenada únicamente en la propia sala, será la sala la responsable de proporcionar dicha información. La sala puede hacer esto mucho mejor que cualquier otro objeto, porque dispone de todo el conocimiento necesario acerca de la estructura interna de almacenamiento de los datos de las salidas. Ahora, dentro de la clase Room, podemos hacer uso del conocimiento de que las salidas están almacenadas en un HashMap, y podemos iterar a través de ese mapa para describirlas.

En consecuencia, sustituimos la versión de getExitString mostrada en el Código 6.6 por la mostrada en el Código 6.7. Este método localiza todos los nombres de las salidas en el HashMap (las claves en el HashMap son los nombres de las salidas) y los concatena en un único objeto de tipo String, que es el que se devuelve. (Necesitamos importar Set de java.util para que esto funcione.)

Ejercicio 6.9 Busque el método keySet en la documentación de HashMap. ¿Qué es lo que hace?

Ejercicio 6.10 Explique en detalle y por escrito cómo funciona el método getExitString mostrado en el Código 6.7.

Código 6.7
Una versión revisada de `getExitString`.

```
/**
 * Devuelve una descripción de las salidas de la sala,
 * por ejemplo, "Exits: north west".
 * @return Una descripción de las salidas disponibles.
 */
public String getExitString()
{
    String returnString = "Exits:";
    Set<String> keys = exits.keySet();
    for(String exit : keys) {
        returnString += " " + exit;
    }
    return returnString;
}
```

Nuestro objetivo de reducir el acoplamiento exige que, en la medida de lo posible, los cambios en la clase `Room` no exijan modificar la clase `Game`. Todavía podemos mejorar este aspecto.

Actualmente, todavía tenemos codificado en la clase `Game` el conocimiento de que la información que deseamos de una sala está compuesta por una cadena de descripción y una cadena que define la salida.

```
System.out.println("You are " + currentRoom.getDescription());
System.out.println(currentRoom.getExitString());
```

¿Qué pasaría si añadiéramos elementos a las salas de nuestro juego? ¿O monstruos? ¿U otros jugadores?

Cuando describimos lo que vemos, la lista de elementos, monstruos y otros jugadores deberían incluirse en la descripción de la sala. No solo necesitaríamos hacer cambios en la clase `Room` para añadir estos elementos, sino que también tendríamos que modificar el segmento de código anterior en el que se imprime la descripción.

Esto es, de nuevo, una violación de la regla del diseño dirigido por responsabilidad. Puesto que la clase `Room` almacena información acerca de una sala, debería ser también ella la que generara una descripción de la sala. Podemos mejorar esto añadiendo a la clase `Room` el siguiente método:

```
/**
 * Devolver una descripción larga de esta sala con el formato:
 *     You are in the kitchen.
 *     Exits: north west
 * @return Un descripción de la sala, incluyendo las salidas.
 */
public String getLongDescription()
{
    return "You are " + description + ".\n" + getExitString();
}
```

En la clase `Game`, escribiremos entonces

```
System.out.println(currentRoom.getLongDescription());
```

La "descripción larga" de una sala incluye ahora la cadena de descripción y la información acerca de las salidas, y en el futuro podría incluir cualquier otra cosa que quisiéramos decir sobre una sala. Cuando hagamos esas ampliaciones futuras, tan solo nos veremos obligados a efectuar cambios en una sola clase: la clase `Room`.

Ejercicio 6.11 Implemente los cambios descritos en esta sección en su propio proyecto *zuul*.

Ejercicio 6.12 Dibuje un diagrama de objetos con todos los objetos de su juego, tal como son justo después de iniciarse el juego.

Ejercicio 6.13 ¿Cómo cambia el diagrama de objetos cuando se ejecuta un comando *go*?

6.8 Localidad de los cambios

Concepto:

Uno de los principales objetivos de un buen diseño de clases es el de conseguir la **localidad de los cambios**: hacer cambios en una clase debería tener un efecto mínimo en las clases restantes.

Otro aspecto de los principios de acoplamiento y responsabilidad es el de la *localidad de los cambios*. Lo que queremos es crear un diseño de clases que facilite los cambios posteriores, haciendo que los efectos de un cambio sean locales.

Idealmente, solo deberíamos tener que cambiar una única clase para realizar una modificación. En ocasiones, será necesario modificar varias clases, pero entonces nuestro objetivo será que se trate del menor número de clases posible. Además, los cambios necesarios en otras clases deberán ser obvios, fáciles de detectar y sencillos de llevar a cabo.

En buena medida, podemos conseguir esto siguiendo unas buenas reglas de diseño, como por ejemplo emplear un diseño dirigido por responsabilidad y tratar de conseguir un acoplamiento débil y una alta cohesión. Además, sin embargo, deberíamos tener en mente las futuras modificaciones y ampliaciones en el momento de crear nuestras aplicaciones. Es importante prever que un cierto aspecto de nuestro programa puede cambiar de cara a hacer que dicho cambio sea lo más fácil posible.

6.9 Acoplamiento implícito

Hemos visto que el uso de campos públicos es una práctica que tiende a crear una forma innecesariamente fuerte de acoplamiento entre clases. Con este acoplamiento fuerte puede ser necesario realizar cambios en más de una clase en aquellas situaciones en las que hubiéramos podido realizar una modificación sencilla. Por tanto, debemos evitar los campos públicos. Sin embargo, existe una forma aun peor de acoplamiento: el *acoplamiento implícito.*

El acoplamiento implícito es una situación en la que una clase depende de la información interna de otra, pero dicha dependencia no es inmediatamente obvia. El acoplamiento fuerte en el caso de los campos públicos no era bueno, pero al menos era obvio: si cambiamos los campos públicos en una clase y nos olvidamos de la otra, la aplicación no podrá compilarse y el compilador nos indicará que hay un problema. Sin embargo, en los casos de acoplamiento implícito, la omisión de un cambio necesario puede no ser detectada.

Podemos ver cómo surge este problema si intentamos añadir palabras de comando adicionales a nuestro juego.

Suponga que queremos añadir el comando *look* (mirar) al conjunto de comandos legales. El propósito de *look* es simplemente imprimir de nuevo la descripción de la sala y las salidas. Esto podría ser útil si hemos introducido una secuencia de comandos en una sala y la descripción se ha desplazado por la pantalla de forma que ya no podemos verla y no nos acordamos de dónde se encuentran las salidas de la sala actual.

Podemos introducir una nueva palabra de comando simplemente añadiéndola a la lista de palabras conocidas en la matriz `validCommands` de la clase `CommandWords`:

```
// Una matriz constante que almacena todas
// las palabras de comando válidas
private static final String validCommands[] = {
    "go", "quit", "help", "look"
};
```

Esto muestra, por cierto, un ejemplo de buena cohesión: en lugar de definir las palabras de comando en el analizador sintáctico, que habría sido una posibilidad obvia, el autor ha creado una clase separada simplemente para definir las palabras de comando. Esto hace que nos resulte muy fácil localizar el lugar en el que están definidas, y también resulta sencillo añadir una nueva. El autor estaba, obviamente, anticipándose a los cambios y asumiendo que posteriormente podrían añadirse más comandos, por lo que creó una estructura para que esa adición sea muy sencilla.

Ya podemos probar la modificación que hemos hecho. Al realizar este cambio y luego ejecutar el juego y escribir el comando `look`, no sucede nada. Esto difiere del comportamiento de la aplicación cuando se encuentra con una palabra de comando desconocida; si escribimos una palabra desconocida, podemos ver la respuesta

```
I don't know what you mean...
```

que indica que la aplicación no entiende lo que hemos escrito. Por tanto, el hecho de que no veamos esta respuesta indica que la palabra ha sido reconocida, pero no sucede nada porque todavía no hemos implementado una acción para este comando.

Podemos corregir esto añadiendo un método para el comando *look* en la clase `Game`:

```
private void look()
{
    System.out.println(currentRoom.getLongDescription());
}
```

También deberíamos, por supuesto, añadir un comentario para este método. Después, tan solo necesitamos añadir un caso para el comando *look* en el método `processCommand`, para invocar el método `look` cuando se reconozca el comando *look*:

```
if(commandWord.equals("help")) {
   printHelp();
}
else if(commandWord.equals("go")) {
    goRoom(command);
}
else if(commandWord.equals("look")) {
    look();
}
```

```
else if(commandWord.equals("quit")) {
    wantToQuit = quit(command);
}
```

Pruebe esta solución y comprobará que funciona.

Ejercicio 6.14 Añada el comando *look* a su versión del juego *zuul*.

Ejercicio 6.15 Añada otro comando a su juego. Para empezar, pruebe a elegir algo simple, como un comando *eat* (comer) que, al ejecutarse, se limite a imprimir "*You have eaten now and you are not hungry any more.*" (¡Ya has comido y ya no tienes más hambre!). Más adelante, podemos mejorar esto de manera que el jugador vaya estando cada vez más hambriento con el paso del tiempo y necesite encontrar comida.

El acoplamiento entre las clases `Game`, `Parser` y `CommandWords` parece hasta ahora ser bastante bueno –fue fácil realizar esta ampliación y hemos conseguido que funcione rápidamente.

El problema mencionado anteriormente –el acoplamiento implícito– se hace patente cuando ahora tratamos de ejecutar el comando *help*. La salida es:

```
You are lost. You are alone. You wander
around at the university.
Your command words are:
  go quit help
```

Analizando la salida, observamos un pequeño problema. El texto de ayuda es incompleto: el nuevo comando *look* no aparece.

Esto parece fácil de corregir: podemos limitarnos a editar la cadena de texto de ayuda en el método `printHelp` de `Game`. Esto se puede hacer rápidamente y no parece ser un mayor problema. Pero suponga que no nos hubiéramos dado cuenta ahora de este error. Piénselo: ¿se le había ocurrido que este problema pudiera presentarse antes de leer estas líneas?

Se trata de un problema fundamental, porque cada vez que añadimos un comando, tenemos que cambiar el texto de ayuda y es bastante sencillo olvidarse de hacer este cambio. El programa se compila y se ejecuta y todo parece ir correctamente. El programador de mantenimiento podría creer que su tarea ha finalizado y lanzar comercialmente un programa que ahora contendrá un error.

Este es un ejemplo de acoplamiento implícito. Cuando los comandos cambian, el texto de ayuda debe modificarse (acoplamiento), pero no hay nada en el código fuente del programa que indique claramente esta dependencia (y es por ello que es implícita).

Una clase bien diseñada evitaría esta forma de acoplamiento siguiendo la regla del diseño dirigido por responsabilidad. Puesto que la clase `CommandWords` es la responsable de las palabras de comando, también debería ser responsable de imprimir la palabras de comando. Por tanto, añadimos el siguiente método a la clase `CommandWords`:

```
/**
 * Imprimir en System.out todos los comandos válidos.
 */
public void showAll()
{
    for(String command : validCommands) {
```

```
            System.out.print(command + " ");
        }
        System.out.println();
    }
```

La idea aquí es que el método `printHelp` de `Game`, en lugar de imprimir un texto fijo con las palabras de comando, invoque un método que pida a la clase `CommandWords` que imprima todas sus palabras de comando. Hacer esto garantiza que siempre se imprimen las palabras de comando correctas, y añadir un nuevo comando hará que también se añada ese comando al texto de ayuda sin necesidad de cambios ulteriores.

El único problema que queda es que el objeto `Game` no tiene una referencia al objeto `CommandWords`. Podemos ver esto en el diagrama de clases (Figura 6.1), donde no aparece ninguna flecha que vaya de `Game` a `CommandWords`. Esto indica que la clase `Game` ni siquiera conoce la existencia de la clase `CommandWords`. En lugar de ello, el juego dispone de un analizador sintáctico, y es este el que tiene palabras de comando.

Ahora podríamos añadir un método al analizador sintáctico que pasara el objeto `CommandWords` al objeto `Game` para que ambos pudieran comunicarse. Sin embargo, esto haría que aumentara el grado de acoplamiento de la aplicación. `Game` dependería entonces de `CommandWords`, mientras que ahora no lo hace. Asimismo, podríamos visualizar este efecto en el diagrama de clases: `Game` tendría entonces una flecha hacia `CommandWords`.

Las flechas en el diagrama son, de hecho, una buena indicación aproximada de lo estrechamente acoplado que está un programa: cuantas más flechas, mayor acoplamiento. Como aproximación a un buen diseño de clases, podemos tratar de crear diagramas que tengan pocas flechas.

Por tanto, el hecho de que `Game` no disponga de una referencia a `CommandWords` es algo bueno. No debemos cambiarlo. Desde el punto de vista de `Game`, el hecho de que la clase `CommandWords` exista es simplemente un detalle de la implementación del analizador sintáctico. El analizador sintáctico devuelve comandos y el que utilice un objeto `CommandWords` para conseguir esto o utilice alguna otra cosa distinta es algo que solo compete a la implementación del analizador sintáctico.

Por tanto, un mejor diseño sería dejar que `Game` hable solo con `Parser`, que a su vez puede hablar con `CommandWords`. Podemos implementar esto añadiendo el siguiente código al método `printHelp` de `Game`:

```
System.out.println("Your command words are:");
parser.showCommands();
```

Lo único que nos queda entonces es escribir el método `showCommands` en `Parser`, que delegará esta tarea a la clase `CommandWords`. A continuación proporcionamos el método completo (en la clase `Parser`):

```
/**
 * Imprimir una lista de las palabras de comando válidas.
 */
public void showCommands()
{
    commands.showAll();
}
```

Ejercicio 6.16 Implemente la versión mejorada del procedimiento de impresión de las palabras de comando, como se describe en esta sección.

Ejercicio 6.17 Si ahora añade otro nuevo comando, ¿seguirá necesitando modificar la clase `Game`? ¿Por qué?

La implementación completa de todos los cambios explicados en el capítulo hasta ahora está disponible en los ejemplos de código en el proyecto denominado *zuul-better*. Si ha hecho los ejercicios hasta aquí, puede ignorar este proyecto y continuar utilizando el suyo propio. Si no ha hecho los ejercicios pero quiere hacer como proyecto de programación los siguientes ejercicios que se presentan en el capítulo, puede utilizar como punto de partida el proyecto *zuul-better*.

6.10 Planificación por adelantado

El diseño que tenemos ahora constituye una mejora importante respecto de la versión original. Sin embargo, es posible mejorarlo aun más.

Una característica de un buen diseñador software es la capacidad de anticiparse a los acontecimientos. ¿Qué cosas se pueden cambiar? ¿Qué cosas podemos suponer, con una cierta garantía, que no sufrirán una modificación durante todo el tiempo de vida del programa?

Una suposición que hemos codificado en la mayoría de nuestras clases es que este juego se ejecutará como un juego basado en texto empleando entrada y salida a través de terminal, pero ¿será siempre así?

Podría ser una ampliación interesante añadir más adelante una interfaz gráfica de usuario con menús, botones e imágenes. En ese caso, ya no querríamos imprimir la información en el terminal de texto. Podemos seguir teniendo palabras de comando y seguir queriendo mostrarlas cuando un jugador introduzca un comando de ayuda. Pero entonces podríamos mostrarlas en un campo de texto dentro de una ventana, en lugar de utilizar `System.out.println`.

Una buena práctica de diseño consiste en tratar de encapsular toda la información acerca de la interfaz de usuario en una única clase, o en un conjunto de clases claramente definido. En nuestra solución de la Sección 6.9, por ejemplo, –el método `showAll` de la clase `CommandWords`– no sigue esta regla de diseño. Sería mucho mejor definir que `CommandWords` es responsable de *generar* (¡pero no de *imprimir*!) la lista de palabras de comando, pero que la clase `Game` debería decidir cómo se presenta esa lista al usuario.

Podemos hacer esto fácilmente modificando el método `showAll`, de modo que devuelva una cadena de caracteres que contenga todas las palabras de comando en lugar de imprimirlas directamente. (Probablemente deberíamos renombrar el método y denominarlo `getCommandList` cuando hagamos este cambio.) Esta cadena puede entonces imprimirse en el método `printHelp` de `Game`.

Observe que esto no nos proporciona ninguna ventaja en este momento, pero este diseño mejorado puede facilitarnos las cosas en el futuro.

Ejercicio 6.18 Implemente el cambio sugerido. Asegúrese de que su programa todavía funciona como antes.

Ejercicio 6.19 Averigüe lo que es el patrón *modelo-vista-controlador* (*model-view-controller*). Puede hacer una búsqueda en la Web para obtener información o puede utilizar cualquier otra fuente de información que localice. ¿Cómo se relaciona ese patrón con el tema que hemos tratado aquí? ¿Qué es lo que sugiere? ¿Cómo se podría aplicar a este proyecto? (Explique solo su aplicación a este proyecto, ya que una implementación real sería un ejercicio avanzado de una gran complejidad.)

6.11 Cohesión

Hemos presentado la idea de la cohesión en la Sección 6.3: cada unidad de código debería ser siempre responsable de una, y solo una, tarea. Ahora vamos a investigar el principio de cohesión en más detalle y a analizar algunos ejemplos.

El principio de cohesión se puede aplicar a clases y métodos: tanto unas como otros deben mostrar un alto grado de cohesión.

6.11.1 Cohesión de métodos

Concepto:

Cohesión de métodos. Un método cohesionado será responsable de una, y solo una, tarea bien definida.

Cuando hablamos de la cohesión de métodos, queremos expresar que, idealmente, cada método debe ser responsable de una, y solo una, tarea bien definida.

Podemos ver un ejemplo de método cohesionado en la clase `Game`. Esta clase tiene un método privado denominado `printWelcome` para mostrar el mensaje de bienvenida, y este método es invocado en el momento de comenzar el juego dentro del método `play` (Código 6.8).

Código 6.8
Dos métodos con un buen grado de cohesión.

```
/**
 * Rutina principal del juego. Ejecuta un bucle hasta
 * el final de la partida
 */
public void play()
{
    printWelcome();

    // Entrar en el bucle principal de comando. En él se leen
    // comandos repetidamente y se ejecutan hasta que
    // el juego finaliza.
    boolean finished = false;
    while (! finished) {
        Command command = parser.getCommand();
        finished = processCommand(command);
    }
    System.out.println("Thank you for playing.  Good bye.");
}
```

Código 6.8
(continuación)
Dos métodos con un buen grado de cohesión.

```
/**
 * Imprimir el mensaje de bienvenida para el jugador.
 */
private void printWelcome()
{
    System.out.println();
    System.out.println("Welcome to The World of Zuul!");
    System.out.println(
          "Zuul is a new, incredibly boring adventure game.");
    System.out.println("Type 'help' if you need help.");
    System.out.println();
    System.out.println(currentRoom.getLongDescription());
}
```

Desde un punto de vista funcional, podríamos habernos limitado a introducir las instrucciones del método `printWelcome` directamente en el método `play`, y habríamos conseguido el mismo resultado sin necesidad de definir un método adicional, ni de realizar una llamada a método. Lo mismo puede decirse, por cierto, del método `processCommand` que también se invoca en el método `play`: también este código podría haberse escrito directamente dentro del método `play`.

Sin embargo, es mucho más fácil entender lo que hace un segmento de código y es también más fácil realizar modificaciones en el mismo, si se utilizan métodos cortos y bien cohesionados. Con la estructura de métodos elegida, todos los métodos son relativamente cortos y fáciles de entender, y sus nombres indican sus propósito de forma bastante clara. Estas características representan una ayuda muy valiosa para el programador de mantenimiento.

6.11.2 Cohesión de clases

Concepto:

Cohesión de clases Una clase cohesionada representa una entidad bien definida.

La regla de la cohesión de las clases afirma que cada clase debería representar una única entidad bien definida dentro del dominio del problema. Como ejemplo de cohesión de clases, vamos a analizar ahora otra ampliación del proyecto *zuul*. Lo que queremos ahora es añadir elementos (*items*) al juego. Cada sala puede tener un elemento y cada elemento tiene una descripción y un peso. El peso de un elemento se puede utilizar posteriormente para determinar si se lo puede coger o no.

Un enfoque simplista sería añadir dos campos a la clase `Room`: `itemDescription` e `itemWeight`. Esto podría funcionar. Con ello, podríamos especificar los detalles del elemento contenido en cada sala y podríamos imprimir esos detalles cada vez que entráramos en una sala.

Sin embargo, este enfoque no presenta un buen grado de cohesión: la clase `Room` ahora describe tanto una sala como un elemento. Esto también sugiere que cada elemento está ligado a una sala concreta, lo que puede que no sea nuestra intención.

Un diseño mejor consistiría en crear una clase separada para los elementos, probablemente denominada `Item`. Esta clase tendría campos para la descripción y el peso, y lo que una sala contendría sería simplemente una referencia a un objeto elemento.

Ejercicio 6.20 Amplíe su proyecto de aventuras o proyecto *zuul-better* para que cada sala solo pueda contener un elemento. Los elementos tienen una descripción y un peso. Al crear salas y definir sus salidas, también hay que crear los elementos para este juego. Cuando un jugador entre en una sala, debe visualizarse la información acerca del elemento presente en la misma, si es que hay uno.

Ejercicio 6.21 ¿Cómo debería generarse la información relativa a un elemento presente en una sala? ¿Qué clase debería generar la cadena de caracteres que describe el elemento? ¿Qué clase debería imprimirla? ¿Por qué? Explíquelo por escrito. Si responder a este ejercicio le hace pensar que debería cambiar su implementación, póngase a ello y haga los cambios.

Las ventajas reales de separar las salas y los elementos dentro del diseño pueden comprobarse si modificamos la especificación ligeramente. En una variante adicional de nuestro juego, queremos permitir que cada sala tenga no solo un único elemento, sino un número ilimitado de elementos. En el diseño que utiliza una clase `Item` separada, esto resulta sencillo: podemos crear múltiples objetos `Item` y almacenarlos en una colección de elementos dentro de la sala.

Con el primer enfoque, más simplista, este cambio habría sido casi imposible de implementar.

Ejercicio 6.22 Modifique el proyecto de modo que una sala pueda albergar cualquier número de elementos. Utilice una colección para hacer esto. Asegúrese de que la sala tenga un métodos `addItem` que permita colocar un elemento en la sala. Cerciórese de que se muestren todos los elementos cuando un jugador entra en una sala.

6.11.3 Cohesión para la legibilidad

Son varias las maneras en que una alta cohesión beneficia a un diseño. Las dos más importantes son la *legibilidad* y la *reutilización*.

El ejemplo visto en la Sección 6.11.1, la cohesión del método `printWelcome`, es claramente un ejemplo en el que incrementar la cohesión hace que una clase sea más legible y, por tanto, más fácil de entender y mantener.

El ejemplo de cohesión de clases de la Sección 6.11.2 también está relacionado con la legibilidad. Si existe una clase separada `Item`, el programador de mantenimiento podrá reconocer fácilmente dónde tiene que comenzar a leer el código, si hace falta realizar una modificación de las características de un elemento. La cohesión de las clases también incrementa la legibilidad de un programa.

6.11.4 Cohesión para la reutilización

La segunda mayor ventaja de la cohesión es que ofrece un mayor potencial de reutilización.

El ejemplo de cohesión de clases de la Sección 6.11.2 también constituye un ejemplo de esto: creando una clase `Item` separada, podemos crear múltiples elementos y usar así el mismo código para más de un elemento.

La reutilización también es un aspecto importante de la cohesión de métodos. Considere un método en la clase `Room` con la siguiente signatura:

```
public Room leaveRoom(String direction)
```

Este método podría devolver la sala en la dirección indicada (para que pueda ser usada como nueva sala actual, `currentRoom`) y también podría imprimir la descripción de la nueva sala en la que acabamos de entrar.

Esto parece un diseño perfectamente factible y podríamos hacer que funcionara. Sin embargo, en nuestra versión, hemos separado esta tarea en dos métodos:

```
public Room getExit(String direction)
public String getLongDescription()
```

El primero de ellos es responsable de devolver la siguiente sala, mientras que el segundo genera la descripción de la sala.

La ventaja de este diseño es que esas tareas separadas pueden reutilizarse más fácilmente. Por ejemplo, el método `getLongDescription` ahora se emplea no solo en el método `goRoom`, sino también en `printWelcome` y en la implementación del comando *look*. Esto solo es posible porque muestra un alto grado de cohesión. Sería imposible reutilizarlo en la versión con el método `leaveRoom`.

Ejercicio 6.23 Implemente un comando *back*. Este comando no tiene segunda palabra. Introducir el comando *back* hace que el jugador vuelva a la sala anterior en la haya estado.

Ejercicio 6.24 Pruebe su nuevo comando. ¿Funciona de la forma esperada? Pruebe también aquellos casos en los que el comando se utilice incorrectamente. Por ejemplo, ¿qué hace su programa si un jugador escribe una segunda palabra detrás del comando *back*? ¿Se comporta de una forma razonable?

Ejercicio 6.25 ¿Qué hace su programa si escribe *back* dos veces? ¿Es este comportamiento razonable?

Ejercicio 6.26 *Ejercicio avanzado* Implemente el comando *back* para que al utilizarlo repetidamente podamos retroceder varias salas, volviendo al principio del juego si usamos el comando suficientes veces. Utilice un objeto `Stack` para hacer esto. (Es posible que tenga que buscar información acerca de estos elementos que implementan pilas *-stacks*. Examine la documentación de la librería Java.)

6.12 Refactorización

Al diseñar aplicaciones, debemos tratar de planificar por adelantado, anticipando los posibles cambios que puedan realizarse en el futuro y creando clases y métodos altamente cohesionados y débilmente acoplados que faciliten las modificaciones. Este es un objetivo muy loable, pero por supuesto no siempre vamos a poder anticipar todas las futuras adaptaciones y tampoco es factible preparase para todas las posibles ampliaciones que podamos imaginar.

Concepto:

La **refactorización** es la actividad consistente en reestructurar un diseño existente para mantener un buen diseño de clases cuando se modifica o amplía la aplicación.

Esta es la razón de que la *refactorización* sea importante.

La refactorización es la actividad consistente en reestructurar las clases y método existentes, para adaptarlos a los cambios en la funcionalidad y en los requisitos. A menudo, a lo largo de la vida de una aplicación se suele ir añadiendo gradualmente funcionalidad. Un efecto común es que, como consecuencia directa, la longitud de los métodos y de las clases va creciendo lentamente.

Resulta tentador para un programador de mantenimiento añadir código adicional a las clases o método existentes. Sin embargo, si seguimos haciendo esto durante un cierto tiempo, el grado de cohesión se reducirá. Cuando se añade más y más código a un método o una clase, es probable que llegue un momento en el que ese método o esa clase representen más de una tarea o entidad claramente definida.

La refactorización consiste en repensar y rediseñar las estructuras de clases y métodos. El efecto más común es que las clases se dividan en dos o que los métodos se dividan en dos o más métodos. La refactorización puede incluir también la unión de varias clases o métodos en uno, aunque esto suele ser bastante menos común que la división.

6.12.1 Refactorización y pruebas

Antes de proporcionar un ejemplo de refactorización, tenemos que reflexionar sobre el hecho de que, al refactorizar un programa, estamos proponiendo normalmente realizar cambios potencialmente grandes en algo que ya funciona. Cuando se modifica algo, hay una cierta probabilidad de que se produzcan errores. Por tanto, es importante actuar con cautela y, antes de refactorizar, debemos asegurarnos de que exista un conjunto de pruebas para la versión actual del programa. Si esas pruebas no existen, entonces debemos decidir primero cómo se puede probar razonablemente la funcionalidad del programa, y dejar constancia de esas pruebas (por ejemplo, anotándolas por escrito) de modo que podamos repetir esas mismas pruebas posteriormente. Hablaremos de las pruebas de una manera más formal en el siguiente capítulo. Si está ya familiarizado con las pruebas automáticas, utilícelas. En caso contrario, unas pruebas manuales (pero sistemáticas) bastan por el momento.

Una vez decidido un conjunto de pruebas, podemos empezar con la refactorización. Idealmente, la refactorización debe realizarse en dos etapas:

- La primera etapa consiste en refactorizar para mejorar la estructura interna del código, pero sin realizar ningún cambio en la funcionalidad de la aplicación. En otras palabras, el programa debería, al ejecutarse, comportarse de la misma forma exacta que antes. Una vez completada esta etapa, habrá que repetir las pruebas previamente decididas para verificar que no hemos introducido errores inadvertidamente.
- La segunda etapa puede acometerse una vez que hayamos restablecido la funcionalidad base en la versión refactorizada. Entonces estaremos en disposición de mejorar el programa. Una vez que lo hayamos hecho, por supuesto, habrá que realizar pruebas con la nueva versión.

Hacer varios cambios al mismo tiempo (refactorizar y añadir nuevas características) hace que sea más difícil localizar las fuentes de error en caso de que introduzcan errores.

Ejercicio 6.27 ¿Qué tipo de pruebas de funcionalidad base podríamos establecer en la versión actual del juego?

6.12.2 Un ejemplo de refactorización

Como ejemplo, vamos a continuar con la ampliación consistente en añadir elementos al juego. En la Sección 6.11.2, hemos comenzado a añadir elementos, sugiriendo una estructura en la que las salas pueden contener cualquier número de elementos. Una ampliación lógica de

esta solución consiste en que el jugador pueda coger elementos y transportarlos. He aquí una especificación informal de nuestro siguiente objetivo:

- El jugador puede coger elementos de la sala actual.
- El jugador puede transportar cualquier número de elementos, pero solo hasta un cierto peso máximo.
- Algunos elementos no pueden cogerse.
- El jugador puede soltar elementos en la sala actual.

Para alcanzar estos objetivos, podemos hacer lo siguiente:

- Si no lo ha hecho ya, añada una clase `Item` al proyecto. Cada elemento tiene, como hemos dicho anteriormente, una descripción (una cadena de caracteres) y un peso (un entero).
- Ahora deberíamos añadir un campo `name` a la clase `Item`, que nos permitirá referirnos al elemento utilizando un nombre que sea más corto que la descripción. Por ejemplo, si hay un libro en la sala actual, los valores de campo de este elemento podrían ser:

name: `book`
description: `an old, dusty book bound in gray leather`
weight: `1200`

Si entramos en una sala, podemos imprimir la descripción del elemento para informar al jugador de qué es lo que hay ahí. Pero para los comandos, el nombre resulta más fácil de utilizar. Por ejemplo, el jugador podría escribir *take book* para coger el libro.

- Podemos garantizar que algunos elementos no puedan cogerse simplemente haciéndolos demasiado pesados (más de lo que un jugador puede transportar). ¿O deberíamos disponer de otro campo de tipo `boolean` que se llamara `canBePickedUp` y que nos indique si se puede coger el elemento? ¿Cuál cree que es el mejor diseño? ¿Tiene alguna importancia? Trate de responder a estas cuestiones pensando en los futuros cambios que puedan efectuarse en el programa.
- Añadiremos comandos *take* y *drop* para coger y soltar elementos. Ambos comandos emplean como segunda palabra un nombre de elemento.
- En algún lugar tenemos que añadir un campo (que almacene algún tipo de colección) para almacenar los elementos que actualmente transporta el jugador. También tenemos que añadir un campo con el peso máximo que el jugador puede transportar, para poder verificarlo cada vez que el jugador trate de coger algo. ¿Dónde debería incluirse ese campo? De nuevo, piense en las futuras ampliaciones de cara a tomar una decisión.

Esta última tarea es la que analizaremos ahora con más detalle para ilustrar el proceso de la refactorización.

La primera pregunta que nos tenemos que plantear al pensar en cómo permitir a los jugadores transportar elementos es: ¿dónde deberíamos añadir los campos para los elementos actualmente transportados y para el peso máximo? Un examen rápido de las clases existentes muestra que la clase `Game` es realmente el único lugar donde encajarían esos campos. No podemos almacenarlos en `Room`, `Item` o `Command`, porque hay muchas instancias distintas de esas clases a lo largo del tiempo, y no todas esas instancias son siempre accesibles. Tampoco tiene sentido incluir esos campos en `Parser` o en `CommandWords`.

Una razón adicional que apoya la decisión de hacer estos cambios en la clase `Game` es el hecho de que esa clase ya almacena la sala actual (la información acerca de dónde se encuentra el jugador en este momento), por lo que añadir los elementos actuales (la información acerca de qué es lo que el jugador tiene) también parece encajar con esto bastante bien.

Esta solución podría funcionar. Sin embargo, no es una solución bien diseñada. La clase `Game` ya es bastante grande y se podría argumentar, con razón, que ya contiene demasiadas cosas. Añadir cosas adicionales no mejora la situación.

Deberíamos preguntarnos de nuevo a qué clase u objeto tendría que pertenecer esta información. Si pensamos despacio en el tipo de información que estamos añadiendo aquí (elementos transportados, peso máximo), nos damos cuenta de que se trata de información sobre un jugador. Lo más lógico (siguiendo las directrices del diseño dirigido por responsabilidad) es crear una clase `Player` para representar al jugador. Entonces podremos añadir estos campos a la clase `Player` y crear un objeto `Player` al principio del juego para almacenar los datos.

El campo existente `currentRoom` también almacena información acerca del jugador: nos indica cuál es la ubicación actual del mismo. En consecuencia, ahora deberíamos mover este campo a la clase `Player`.

Analizando esta solución, resulta obvio que este diseño encaja mejor con el principio del diseño dirigido por responsabilidad. ¿Quién debería ser responsable de almacenar la información acerca del jugador? Por supuesto, la clase `Player`.

En la versión original, sólo disponíamos de un elemento de información relativo al jugador: la sala actual. Podría discutirse si aun así hubiéramos debido tener una clase `Player`. Habría razones tanto a favor como en contra. Se habría tratado de un diseño elegante, así que tal vez hubiéramos debido definir esa clase. Pero tener una clase con un único campo y ningún método que haga nada significativo podría considerarse un desperdicio.

En ocasiones, hay áreas de sombra como esta en las que podría defenderse una decisión o la contraria. Pero después de añadir estos nuevos campos, la situación es muy clara. Ahora existe un buen argumento en favor de definir una clase `Player`. Esta clase se encargaría de almacenar los campos y dispondría de métodos como `dropItem` y `pickUpItem` para soltar y coger elementos (métodos que podrían incluir la comprobación del peso y devolver el valor *false* si no podemos transportar ese elemento).

Lo que hemos hecho al introducir la clase `Player` y mover el campo `currentRoom` de `Game` a `Player` es una refactorización. Hemos reestructurado la forma en que representamos nuestros datos, para conseguir un mejor diseño con unos requisitos modificados.

Los programadores peor formados que nosotros (o simplemente perezosos) podrían haber dejado el campo `currentRoom` donde estaba, al ver que el programa funciona tal como está y que no parece que exista una necesidad imperiosa de realizar esta modificación. Como resultado, terminarían teniendo un diseño de clases muy lioso.

Podemos ver el efecto de realizar esta modificación si tratamos de anticiparnos aún más a los acontecimientos. Suponga que ahora quisiéramos ampliar el juego para permitir la existencia de múltiples jugadores. Con nuestro nuevo y elegante diseño, esto pasa a ser muy sencillo. Ya disponemos de una clase `Player` (el objeto `Game` almacena un objeto `Player`), y es fácil crear varios objetos `Player` y almacenar en `Game` una colección de jugadores, en lugar de un único jugador. Cada objeto jugador almacenará ahora su sala actual, sus elementos y su peso máximo. Los diferentes jugadores podrían incluso tener distintos pesos máximos, abriendo la

puerta al concepto, aun más amplio, de disponer de jugadores con capacidades muy distintas –siendo su capacidad de transportar elementos simplemente una más de las múltiples posibilidades que se nos pueden ocurrir.

Sin embargo, el programador perezoso que hubiera dejado `currentRoom` en la clase `Game` ahora tendría un serio problema. Puesto que el juego completo dispone de una única sala actual, las ubicaciones actuales de los múltiples jugadores no pueden almacenarse fácilmente. Los malos diseños suelen pasarnos la factura posteriormente, y terminan dándonos más trabajo.

El llevar a cabo una buena refactorización tiene tanto que ver con la capacidad de pensar de una manera determinada, como con nuestras habilidades técnicas. Mientras realizamos cambios y ampliaciones en una aplicación, debemos preguntarnos periódicamente si el diseño original de las clases sigue representando la mejor solución. A medida que cambia la funcionalidad, van variando los argumentos a favor o en contra de ciertos diseños. Lo que era un buen diseño para una aplicación simple, puede dejar de serlo al añadir ciertas extensiones.

Reconocer estos cambios y llevar a cabo la refactorización del código fuente suele terminar ahorrando una gran cantidad de tiempo y esfuerzo. Por regla general, cuando antes limpiemos nuestro diseño, más trabajo nos ahorraremos.

Debemos estar preparados para *desgajar* métodos (transformar una secuencia de instrucciones del cuerpo de un método existente en un método nuevo e independiente) y clases (tomar partes de una clase y crear otra nueva a partir de ellas). Tomar en consideración de manera periódica la refactorización hace que nuestro diseño de clases siga siendo limpio y nos acaba ahorrando trabajo al final. Por supuesto, una de las cosas que hará que la refactorización nos complique la vida a largo plazo es que no probemos adecuadamente la versión refactorizada comparándola con la original. Cada vez que nos embarquemos en una tarea de refactorización de una cierta envergadura resulta esencial garantizar que se prueben las cosas adecuadamente, antes y después del cambio. Hacer esas pruebas manualmente (creando y probando objetos interactivamente) puede volverse tedioso muy rápidamente. En el siguiente capítulo investigaremos cómo se pueden mejorar las pruebas por el procedimiento de automatizarlas.

Ejercicio 6.28 Refactorice su proyecto para introducir una clase `Player` separada. Cada objeto `Player` debe almacenar al menos la sala actual del jugador, pero si quiere puede hacer que almacene también el nombre del jugador o alguna otra información.

Ejercicio 6.29 Implemente una extensión que permita a un jugador coger un único elemento. Esto incluye implementar dos nuevos comandos: *take* y *drop*.

Ejercicio 6.30 Amplíe su implementación para permitir que el jugador transporte cualquier número de elementos.

Ejercicio 6.31 Añada una restricción que permita al jugador transportar elementos solo hasta un peso máximo especificado. El peso máximo que cada jugador puede transportar es un atributo del jugador.

Ejercicio 6.32 Implemente un comando *items* que haga que se impriman todos los elementos actualmente transportados, junto con su peso total.

Ejercicio 6.33 Añada un elemento *magic cookie* (galleta mágica) a una sala. Añada un comando *eat cookie* (comer galleta). Si un jugador encuentra la galleta mágica y se la come, se incrementará el peso que el jugador puede transportar. (Si quiere puede modificar esto ligeramente para que encaje mejor en su propio escenario de juego.)

6.13 Refactorización para la independencia respecto del idioma

Una característica del juego *zuul* que todavía no hemos comentado es que la interfaz del usuario está estrechamente ligada a comandos escritos en inglés. Esta suposición está integrada tanto en la clase `CommandWords`, en la que se almacenan los comandos válidos, como en la clase `Game`, en la que el método `processCommand` compara explícitamente cada palabra de comando con un conjunto de palabras en inglés. Si queremos cambiar la interfaz para permitir a los usuarios utilizar un idioma distinto, entonces tendremos que localizar todos los lugares del código fuente en los que se utilizan palabras de comando y modificar esas secciones de código. Este es un ejemplo más de un acoplamiento implícito, concepto que hemos explicado en la Sección 6.9.

Si queremos que el programa sea independiente del idioma, entonces deberíamos tener, idealmente, un único lugar en el código fuente en el que se almacene el texto real de las palabras de comando, y hacer que en todos los demás lugares se haga referencia a los comandos en un forma que sea independiente del idioma. Una característica del lenguaje de programación que hace que esto sea posible son los *tipos enumerados* o *enums*. Exploraremos esta característica de Java mediante los proyectos *zuul-with-enums*.

6.13.1 Tipos enumerados

El Código 6.9 muestra la definición de un tipo enumerado Java denominado `CommandWord`.

En su forma más simple, una definición de tipo enumerado consta de un envoltorio exterior que utiliza la palabra `enum` en lugar de `class` y de un cuerpo que es simplemente una lista de

Código 6.9
Un tipo enumerado Java para palabras de comando.

```
/**
 * Representaciones para todas las palabras de
 * comando válidas del juego.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 2011.08.09
 */
public enum CommandWord
{
    // Un valor para cada palabra de comando, más otro
    // para los comandos no reconocidos.
    GO, QUIT, HELP, UNKNOWN;
}
```

nombres variables, que indica el conjunto de valores pertenecientes a este tipo. Por convenio, estos nombres variables se escriben siempre en mayúsculas. En los programas, nunca crearemos objetos de tipo enumerado. De hecho, cada nombre contenido en la definición del tipo representa una instancia diferente de ese tipo que ya ha sido creada para que la podamos emplear. Haremos referencia a esas instancias mediante `CommandWord.GO`, `CommandWord.QUIT`, etc. Aunque la sintaxis para utilizarlas es similar, es importante que evitemos pensar en estos valores como si fueran parecidos a las constantes de clase numérica que hemos visto en la Sección 5.13. A pesar de la simplicidad de su definición, los valores de tipo enumerado son realmente objetos, y no equivalen a números enteros.

¿Cómo podemos utilizar el tipo `CommandWord` como ayuda para intentar desacoplar la lógica del juego de *zuul* con respecto a cualquier idioma concreto? Una de las primera mejoras que podemos hacer es la siguiente serie de comprobaciones dentro del método `processCommand` de `Game`:

```
if(command.isUnknown()) {
    System.out.println("I don't know what you mean...");
    return false;
}
String commandWord = command.getCommandWord();
if(commandWord.equals("help")) {
    printHelp();
}
else if(commandWord.equals("go")) {
    goRoom(command);
}
else if(commandWord.equals("quit")) {
    wantToQuit = quit(command);
}
```

Si hacemos que `commandWord` sea de tipo `CommandWord` en lugar de ser de tipo `String`, entonces podemos reescribir esto como:

```
if(commandWord == CommandWord.UNKNOWN) {
   System.out.println("I don't know what you mean...");
}
else if(commandWord == CommandWord.HELP) {
   printHelp();
}
else if(commandWord == CommandWord.GO) {
   goRoom(command);
}
else if(commandWord == CommandWord.QUIT) {
   wantToQuit = quit(command);
}
```

De hecho, ahora que hemos cambiado el tipo a `CommandWord`, también podríamos utilizar una *instrucción switch* en lugar de la serie de instrucciones if. Esto expresa las intenciones de este segmento de código con algo más de claridad.[2]

[2] En Java 7, las cadenas de caracteres también pueden utilizarse como valores en las instrucciones switch. En Java 6 y versiones anteriores, las cadenas de caracteres no pueden emplearse instrucciones en switch.

```
switch (commandWord) {
    case UNKNOWN:
        System.out.println("I don't know what you mean...");
        break;
    case HELP:
        printHelp();
        break;
    case GO:
        goRoom(command);
        break;
    case QUIT:
        wantToQuit = quit(command);
        break;
}
```

Concepto:

Una **instrucción switch** selecciona una secuencia de instrucciones para su ejecución a partir de una serie de múltiples opciones diferentes.

La instrucción switch toma la variable encerrada entre los paréntesis que siguen a la palabra clave `switch` (en nuestro caso, `commandWord`) y la compara con cada uno de los enumerados detrás de las palabras clave `case`. Cuando se encuentra una correspondencia, se ejecuta el código situado detrás de ella. La instrucción `break` hace que la ejecución de la instrucción `switch` se interrumpa en dicho punto y que la ejecución continúe después de dicha instrucción `switch`. Para ver una descripción más completa de la instrucción switch, consulte el Apéndice D.

Ahora simplemente tenemos que hacer que los comandos escritos por el usuario se asignen a los valores correspondientes de `CommandWord`. Abra el proyecto *zuul-with-enums-v1* para ver cómo hemos hecho esto. El cambio más significativo se encuentra en la clase `CommandWords`. En lugar de usar una matriz de cadena de caracteres para definir los comandos válidos, ahora usamos un mapa que establece correspondencias entre cadenas de caracteres y objetos `CommandWord`:

```
public CommandWords()
{
    validCommands = new HashMap<String, CommandWord>();
    validCommands.put("go", CommandWord.GO);
    validCommands.put("help", CommandWord.HELP);
    validCommands.put("quit", CommandWord.QUIT);
}
```

El comando escrito por un usuario puede ahora convertirse fácilmente a su valor correspondiente de tipo enumerado.

Ejercicio 6.34 Repase el código fuente del proyecto *zuul-with-enums-v1* para ver cómo utiliza el tipo `CommandWord`. Se han adaptado las clases `Command`, `CommandWords`, `Game` y `Parser` de la versión *zuul-better* para reflejar este cambio. Compruebe que el programa sigue funcionando como cabría esperar.

Ejercicio 6.35 Añada un comando *look* al juego, según las ideas descritas en la Sección 6.9.

Ejercicio 6.36 "Traduzca" el juego para utilizar diferentes palabras de comando para los comandos `GO` y `QUIT`. Puede tratarse de palabras de un idioma real o de palabras que usted mismo se invente. ¿Basta con editar la clase `CommandWords` para que este cambio funcione? ¿Cuál es la importancia de esto?

Ejercicio 6.37 Cambie la palabra asociada con el comando HELP y compruebe que funciona correctamente. Después de haber realizado los cambios, ¿qué podemos observar en el mensaje de bienvenida que se imprime cuando comienza el juego?

Ejercicio 6.38 En un nuevo proyecto, defina su propio tipo enumerado denominado `Position` (posición) con valores `TOP`, `MIDDLE` y `BOTTOM`.

6.13.2 Desacoplamiento adicional de la interfaz de comandos

El tipo enumerado `CommandWord` nos ha permitido desacoplar significativamente el idioma de la interfaz de usuario con respecto a la lógica del juego, y es posible casi completamente traducir los comandos a otro idioma simplemente editando la clase `CommandWords`. (En algún momento, deberemos también traducir las descripciones de las salas y otras cadenas de salida, probablemente leyéndolas de un archivo, pero dejaremos esto para más adelante.) Hay una tarea adicional de desacoplamiento de las palabras de comando que nos gustaría realizar. Actualmente, cada vez que se introduce un nuevo comando en el juego, tenemos que añadir un nuevo valor a `CommandWord` y tenemos que establecer una asociación entre el valor y el texto del usuario en las clases `CommandWords`. Sería bastante útil que pudiéramos hacer que el tipo `CommandWord` fuera autocontenido –moviendo la asociación texto:valor de `CommandWords` a `CommandWord`.

Java permite que las definiciones de tipos enumerados contengan mucho más que una lista de los valores del tipo. No vamos a explorar esta característica con demasiado detalle, pero sí que daremos alguna indicación de qué cosas son posibles. El Código 6.10 muestra un tipo `CommandWord` mejorado que tiene un aspecto bastante similar a la definición de una clase normal. Puede encontrarlo en el proyecto *zuul-with-enums-v2*.

Código 6.10
Asociación de cadenas de comandos con valores de un tipo enumerado.

```
/**
 * Representaciones para todas las palabras de comando válidas
 * para el juego, junto con una cadena de caracteres
 * en un idioma concreto.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 2011.08.10
 */
public enum CommandWord
{
    // Un valor para cada palabra de comando, junto con
    // su correspondiente cadena de caracteres para la
    // interfaz de usuario.
    GO("go"), QUIT("quit"), HELP("help"), UNKNOWN("?");

    // La cadena de caracteres del comando.
    private String commandString;
```

Código 6.10
(*continuación*)
Asociación de cadenas de comandos con valores de un tipo enumerado.

```
    /**
     * Inicializar con la correspondiente cadena de
     * de caracteres comando.
     * @param commandString Cadena de comando.
     */
    CommandWord(String commandString)
    {
        this.commandString = commandString;
    }

    /**
     * @return La palabra de comando en forma de cadena.
     */
    public String toString()
    {
        return commandString;
    }
}
```

Los puntos principales que hay que resaltar en esta nueva versión de `CommandWord` son:

- Cada valor del tipo va seguido por un valor de parámetro; en este caso, el texto del comando asociado con dicho valor.
- La definición del tipo incluye un constructor. Este no tiene la palabra `public` en su cabecera. Los constructores de los tipos enumerados nunca son públicos, puesto que no somos nosotros quienes creamos las instancias. El parámetro asociado con cada valor del tipo se pasa a este constructor.
- La definición del tipo incluye un campo, `commandString`. El constructor almacena la cadena de comando en este campo.
- Se ha utilizado un método `toString` para devolver el texto asociado con cada valor del tipo.

Con el texto de los comandos almacenado en el tipo `CommandWord`, la clase `CommandWords` en *zuul-with-enums-v2* utiliza una forma distinta para crear su mapa que establece la asignación entre el texto y los valores enumerados:

```
validCommands = new HashMap<String, CommandWord>();
for(CommandWord command : CommandWord.values()) {
    if(command != CommandWord.UNKNOWN) {
        validCommands.put(command.toString(), command);
    }
}
```

Todo tipo enumerado define un método `values` que devuelve una matriz rellena con los objetos correspondientes a los valores del tipo. El código anterior itera a través de la matriz y llama al método `toString` para obtener el objeto `String` correspondiente al comando asociado con cada valor.

Ejercicio 6.39 Añada su propio comando *look* a *zuul-with-enums-v2*. ¿Solo hace falta modificar el tipo `CommandWord`?

Ejercicio 6.40 Cambie la palabra asociada con el comando *help* en `CommandWord`. ¿Se ve reflejado este cambio automáticamente en el mensaje de bienvenida que aparece al comenzar el juego? Examine el método `printWelcome` de la clase `Game` para ver cómo se ha conseguido esto.

6.14 Directrices de diseño

Un consejo muy repetido a los principiantes con el fin de escribir buenos programas orientados a objetos es el siguiente: "No incluyas demasiadas cosas en un mismo método" o "No metas todo en una única clase". Ambas sugerencias tienen sentido, pero frecuentemente conducen a plantearse la cuestión contraria: "¿Qué longitud debe tener un método? "o "¿Qué longitud debe tener una clase?".

Después de las explicaciones dadas en este capítulo, estas cuestiones pueden ahora responderse en términos de los conceptos de cohesión y de acoplamiento. Un método será demasiado largo si lleva a cabo más de una tarea lógica. Una clase será demasiado compleja si representa más de una entidad lógica.

Observará que estas respuestas no proporcionan reglas claras y precisas que especifiquen exactamente qué es lo que tenemos que hacer. Conceptos tales como *una tarea lógica* siguen estando abiertos a la interpretación y los distintos programadores tomarán decisiones diferentes en muchas situaciones.

Se trata de *directrices* (que no están grabadas en piedra). No obstante, recordar esas directrices le permitirá mejorar significativamente el diseño de sus clases y le permitirá también dominar problemas más complejos y escribir programas mejores y más interesantes.

Es importante comprender que los siguientes ejercicios son sugerencias, no especificaciones fijas. Este juego tiene múltiples formas en las que podría ser ampliado, y le animamos a inventar sus propias extensiones. No es necesario realizar todos los ejercicios que aquí se detallan para crear un juego interesante; puede que quiera llevar a cabo más ampliaciones u otras distintas. He aquí, algunas sugerencias con las que puede empezar.

Ejercicio 6.41 Añada alguna forma de límite temporal a su juego. Si una determinada tarea no se completa en el tiempo especificado, el jugador pierde. Un límite de tiempo puede implementarse fácilmente contando el número de movimientos o el número de comandos que se haya introducido. No es necesario utilizar tiempo real.

Ejercicio 6.42 Implemente una puerta secreta en alguna parte (o algún otro tipo de puerta que solo pueda cruzarse en una dirección).

Ejercicio 6.43 Añada un *beamer* (teletransportador de rayos) al juego. El teletransportador de rayos es un dispositivo que puede cargarse (*charge*) y dispararse (*fire*). Al cargar el teletransportador, este memoriza la

sala actual. Cuando se dispara, lleva de vuelta al jugador de forma inmediata a la sala en la que fue cargado. El teletransportador de rayos puede ser un equipamiento o un elemento que el jugador puede encontrar. Por supuesto, tendrá que introducir comandos para cargar y disparar el teletransportador.

Ejercicio 6.44 Añada puertas cerradas a su juego. El jugador necesitará encontrar (u obtener de alguna manera) una llave para abrir una puerta.

Ejercicio 6.45 Añada una sala transportadora. Cuando el jugador entre en esta sala será transportado aleatoriamente a una de las otras salas. Nota: desarrollar un diseño adecuado para esta tarea no es trivial. Puede que sea interesante que discuta las alternativas de diseño con otros estudiantes. (Hablaremos de las alternativas de diseño para esta tarea al final del Capítulo 9. El lector avanzado o dado a la experimentación puede saltar a ese capítulo y echar un vistazo.)

Ejercicio 6.46 *Ejercicio avanzado* En el método `processCommand` de `Game`, hay un instrucción switch (o una secuencia instrucciones if) para efectuar la tarea de despacho de los comandos, cuando se reconoce una palabra de comando. No es un diseño muy elegante, porque cada vez que añadimos un comando, hay que añadir un nuevo caso ahí. ¿Podría mejorar este diseño? Diseñe las clases para que el manejo de los comandos sea más modular y para que los comandos nuevos se puedan añadir más fácilmente. Implemente su solución y pruébela.

Ejercicio 6.47 Añada personajes al juego. Los personajes son similares a los elementos, pero pueden hablar. La primera vez que nos encontramos con ellos, pronuncian un cierto texto, y pueden proporcionar al jugador algo de ayuda si este les da el elemento correcto.

Ejercicio 6.48 Añada personajes en movimiento. Son como los otros personajes, pero cada vez que el jugador escribe un comando, estos personajes se pueden desplazar a una sala adyacente.

6.15 Ejecución sin BlueJ

Cuando nuestro juego esté terminado, es posible que queramos pasárselo a otras personas para que jueguen. Para ello, sería conveniente que la gente pudiera jugar sin necesidad de iniciar BlueJ. Para poder hacer esto, necesitamos una cosa más: los *métodos de clase,* que en Java también se denominan *métodos estáticos*.

6.15.1 Métodos de clase

Hasta ahora, todos los métodos que hemos visto eran *métodos de instancia*: se invocan sobre una instancia de una clase. Lo que distingue a los métodos de clase de los métodos de instancia es que los primeros pueden invocarse sin una instancia –basta con disponer de la clase.

En la Sección 5.13 hemos hablado de las variables de clase. Los métodos de clase están relacionados conceptualmente con ellas y utilizan una sintaxis relacionada (la palabra clave `static` en Java). Al igual que las variables de clase pertenecen a la clase y no a una instancia, también es así en el caso de los métodos de clase.

Un método de clase se diseña añadiendo la palabra clave `static` delante del nombre del tipo en la signatura del método:

```
public static int getNumberOfDaysThisMonth()
{
    ...
}
```

Dicho método puede entonces invocarse especificando el nombre de la clase en la que está definido antes del punto en la notación con punto usual. Por ejemplo, si definimos el método anterior en una clase denominada `Calendar`, la siguiente llamada permitiría invocarlo:

```
int days = Calendar.getNumberOfDaysThisMonth();
```

Observe que antes del punto se utiliza el nombre de la clase –no se ha creado ningún objeto.

Ejercicio 6.49 Lea la documentación de la clase `Math` en el paquete `java.lang`. Contiene muchos métodos estáticos. Localice el método que calcula el máximo de dos números enteros. ¿Cuál es su signatura?

Ejercicio 6.50 ¿Por qué cree que los métodos de la clase `Math` son estáticos? ¿Podrían escribirse como métodos de instancias?

Ejercicio 6.51 Escriba una clase de prueba que disponga de un método para comprobar cuánto tarda en contarse desde 1 hasta 100 en un bucle. Puede utilizar el método `currentTimeMillis` de la clase `System` como ayuda para medir el tiempo.

6.15.2 El método `main`

Si queremos iniciar una aplicación Java sin BlueJ, necesitamos utilizar un método de clase. En BlueJ, lo que hacemos normalmente es crear un objeto e invocar uno de sus métodos, pero sin BlueJ, la aplicación se iniciará sin que exista ningún objeto. Lo único que tenemos inicialmente son las clases, por lo que el primer método que pueda invocarse debe ser un método de clase.

La solución Java para iniciar aplicaciones es muy simple: el usuario especifica la clase que debe iniciarse y el sistema Java invocará entonces un método denominado `main` en dicha clase. Este método debe disponer de una signatura específica. Si no existe dicho método en esa clase, el sistema informará de que se ha producido un error. En el Apéndice E se describen los detalles de este método y los comandos necesarios para iniciar el sistema Java sin BlueJ.

Ejercicio 6.52 Localice los detalles del método `main` y añada dicho método a su clase `Game`. El método debe crear un objeto `Game` e invocar el método `play` sobre él. Pruebe el método `main` invocándolo desde BlueJ. Los métodos de clase pueden invocarse desde el menú emergente de la clase.

Ejercicio 6.53 Ejecute su juego sin BlueJ.

6.15.3 Limitaciones de los métodos de clase

Puesto que los métodos de clase están asociados con una clase en lugar de con una instancia, tienen dos limitaciones importantes. La primera de ellas es que un método de clase no puede acceder a ningún campo de instancia definido en la clase. Esto es lógico porque los campos de instancia están asociados con objetos individuales. En lugar de ello, los métodos de clase están restringidos y solo pueden acceder a las variables de clase definidas dentro de su clase. La segunda limitación se parece a la primera: un método de clase no puede invocar ningún método de instancia de esa clase. Un método de clase solo puede invocar otros métodos de clase definidos dentro de su clase.

Como comprobará si sigue leyendo, en los ejemplos de este libro vamos a hacer muy poco uso de los métodos de clase.

6.16 Resumen

En este capítulo, hemos hablado de lo que a menudo se denominan *aspectos no funcionales* de una aplicación. En lo que a ellos respecta, el problema no es tanto hacer que un programa realice una cierta tarea, sino hacerlo mediante clases bien diseñadas.

Un buen diseño de clases puede implicar una enorme diferencia a la hora de corregir, modificar o ampliar una aplicación. También nos permite reutilizar partes de la aplicación en otros contextos (por ejemplo, para otros proyectos) y presenta, por tanto, ventajas futuras.

Hay dos conceptos clave que permiten evaluar el diseño de las clases: el acoplamiento y la cohesión. El acoplamiento hace referencia a la interconexión entre las clases, mientras que la cohesión se refiere a la modularización en unidades apropiadas. Un buen diseño exhibe un acoplamiento débil y una alta cohesión.

Una forma de conseguir una buena estructura es seguir un proceso de diseño dirigido por responsabilidad. Cada vez que añadamos una funcionalidad a la aplicación, intentaremos identificar qué clase debe ser responsable de cada parte de la tarea.

A la hora de ampliar un programa, utilizamos periódicamente la refactorización para adaptar el diseño a los cambios en los requisitos y para garantizar que las clases y los métodos continúen estando cohesionados y sigan estando débilmente acoplados.

Términos introducidos en el capítulo

duplicación de código, acoplamiento, cohesión, encapsulación, diseño dirigido por responsabilidad, acoplamiento implícito, refactorización, método de clase

Resumen de conceptos

- **acoplamiento** El término acoplamiento describe la interconexión de las clases. Lo que buscamos en un sistema es un acoplamiento débil; es decir, un sistema en el que cada clase sea fundamentalmente independiente y se comunique con las otras clases a través de una interfaz compacta y bien definida.

- **cohesión** El término cohesión describe lo bien que una unidad de código se corresponde con una tarea lógica o con una entidad. En un sistema muy cohesionado, cada unidad de código (método, clase o módulo) es responsable de una tarea o entidad bien definidas. Un buen diseño de clases exhibe un alto grado de cohesión.
- **duplicación de código** La duplicación de código (tener el mismo segmento de código en una aplicación más de una vez) es un signo de mal diseño. Se debe intentar evitar.
- **encapsulación** Una adecuada encapsulación de las clases reduce el acoplamiento y conduce, por tanto, a un mejor diseño.
- **diseño dirigido por responsabilidad** El diseño dirigido por responsabilidad es el proceso de diseñar clases asignando unas responsabilidades bien definidas a cada clase. Este proceso puede emplearse para determinar qué clase debería implementar cada parte de una función de la aplicación.
- **localidad de los cambios** Uno de los principales objetivos de un buen diseño de clases es el de conseguir la localidad de los cambios: hacer cambios en una clase debería tener un efecto mínimo en las clases restantes.
- **cohesión de métodos** Un método cohesionado será responsable de una, y solo una, tarea bien definida.
- **cohesión de clases** Una clase cohesionada representa una entidad bien definida.
- **refactorización** La refactorización es la actividad consistente en reestructurar un diseño existente para mantener un buen diseño de clases cuando se modifica o amplía la aplicación.
- **instrucción switch** Una instrucción switch selecciona una secuencia de instrucciones para su ejecución a partir de una serie de múltiples opciones diferentes.

Ejercicio 6.54 Sin utilizar BlueJ, edite su proyecto *TechSupport* del Capítulo 5 de modo que pueda ejecutarse sin BlueJ. Después, ejecútelo desde la línea de comandos.

Ejercicio 6.55 ¿Se puede invocar un método estático desde un método de instancia? ¿Se puede invocar un método de instancia desde un método estático? ¿Se puede invocar un método estático desde un método estático? Responda por escrito a estas preguntas y luego cree un proyecto de prueba para comprobar sus respuestas y probarlas. Explique detalladamente sus respuestas y observaciones.

Ejercicio 6.56 ¿Puede una clase llevar la cuenta de cuántas instancias de esa clase se han creado? ¿Qué hace falta para ello? Escriba algunos fragmentos de código que ilustren lo que hay que hacer. Suponga que quiere un método estático denominado `numberOfInstances` que devuelva el número de instancias creadas.

CAPÍTULO

7

Objetos con un buen comportamiento

Principales conceptos explicados en el capítulo:

- pruebas
- prueba de unidades
- depuración
- automatización de las pruebas

Estructuras Java explicadas en este capítulo:

(En este capítulo no se presenta ninguna nueva estructura Java.)

7.1 Introducción

Si ha leído los capítulos anteriores del libro y ha implementado los ejercicios que en ellos sugeríamos, entonces habrá escrito al momento un buen número de clases. Una observación que probablemente haya hecho es que cualquier clase que escriba raramente es perfecta después del primer intento de escribir su código fuente. Normalmente, al principio no funciona correctamente, y es necesario algo más de trabajo para completarla.

Los problemas con los que tenga que enfrentarse irán cambiando a lo largo del tiempo. Los principiantes suelen tener que lidiar con los *errores sintácticos* de Java. Los errores sintácticos son errores en la estructura del propio código fuente. Son fáciles de localizar porque el compilador los resaltará y mostrará algún tipo de mensaje de error.

Los programadores con más experiencia que afronten problemas más complicados suelen tener menos dificultades con la sintaxis del lenguaje. En lugar de ello, lo que más les afecta son los *errores lógicos*.

Un error lógico es un problema en el que el programa se compila y se ejecuta sin que se produzca ningún error obvio, pero nos proporciona el resultado incorrecto. Los problemas lógicos son mucho más difíciles de localizar que los errores de sintaxis. De hecho, en ocasiones no es fácil detectar siquiera que existe un error.

Es relativamente fácil aprender a escribir programas sintácticamente correctos, y existen buenas herramientas (como los compiladores) para detectar y señalar los errores sintácticos. Por el contrario, escribir programas lógicamente correctos es muy difícil para cualquier problema no trivial, y la prueba de que un programa es correcto no puede automatizarse, por

regla general. Es tan difícil, de hecho, que sabemos que la mayor parte del software que se vende comercialmente contiene un número significativo de errores.

Concepto:

Las **pruebas** son la actividad consistente en averiguar si un fragmento de código (un método, una clase o un programa) presenta el comportamiento deseado.

Por tanto, es esencial para un ingeniero software competente aprender a tratar con los asuntos de corrección de los programas y saber cómo reducir el número de errores existentes en una clase.

En este capítulo, vamos a hablar de diversas actividades que están relacionadas con la tarea de mejora de la corrección de un programa. Dichas tareas incluyen las pruebas, la depuración y la escritura de programas con vistas a su mantenibilidad.

Las *pruebas* son una actividad que se preocupa de averiguar si un segmento de código contiene errores. Realizar unas buenas pruebas no es sencillo, y son muchos los aspectos que hay que tener en cuenta a la hora de probar un programa.

Concepto:

La **depuración** es el intento de localizar y corregir el origen de un error.

La *depuración* viene después de las pruebas. Si las pruebas indican que existe algún error, utilizamos técnicas de depuración para averiguar dónde está exactamente el error y corregirlo. Entre el momento de saber que existe un error y el de localizar la causa y corregirlo, puede requerirse una cantidad significativa de trabajo.

La *escritura de programas para su mantenibilidad* es tal vez el tema más fundamental. Se trata de intentar de escribir código de tal manera de que se eviten en primer lugar los errores y que si, a pesar de todo, se producen, puedan ser encontrados lo más fácilmente posible. El estilo de codificación y los comentarios son parte de esa tarea, como también lo son los principios de calidad del código de los que hemos hablado en el capítulo anterior. Idealmente, el código debería ser fácil de entender, para que el programador original evite introducir errores y el programador de mantenimiento pueda localizar los posibles errores fácilmente.

En la práctica, esto no siempre es simple. Pero existe una enorme diferencia entre tener unos pocos y tener muchos errores, como también existe una enorme diferencia entre el esfuerzo que requiere depurar código bien escrito y el que hace falta para el código que no ha sido escrito con tanto cuidado.

7.2 Pruebas y depuración

Las pruebas y la depuración son habilidades cruciales en el desarrollo de software. A menudo necesitará comprobar sus programas en busca de errores y luego localizar la fuente de esos errores cuando aparezcan. Además, puede que también le toque responsabilizarse de probar los programas de otras personas o de modificarlos. En este último caso, la tarea de depuración está estrechamente relacionada con el proceso de comprender el código escrito por algún otro programador, y hay un alto grado de solapamiento en las técnicas que pueden aplicarse a ambas tareas. En las próximas secciones, estudiaremos las siguientes técnicas de pruebas y depuración:

- Prueba manual de unidades dentro de BlueJ.
- Automatización de pruebas.
- Recorridos manuales.
- Instrucciones de impresión.
- Depuradores.

Examinaremos las dos primeras técnicas de prueba en el contexto de algunas clases que podríamos haber escrito para nosotros mismos, mientras que las técnicas restantes las analizaremos en el contexto de la tarea de intentar comprender el código fuente escrito por alguna otra persona.

7.3 Prueba de unidades dentro de BlueJ

El término *prueba de unidades* hace referencia a una prueba de las partes individuales de una aplicación, por oposición a las *pruebas de la aplicación*, que consisten en probar la aplicación completa. Las unidades sometidas a prueba pueden ser de diversos tamaños. Pueden ser un grupo de clases, una única clase o incluso un único método. Merece la pena observar que la prueba de unidades puede tener lugar mucho antes de completar una aplicación. Cualquier método puede (y debe) ser probado después de haber sido escrito y compilado.

Puesto que BlueJ nos permite interactuar directamente con objetos individuales, nos ofrece formas originales de realizar las pruebas de las clases y los métodos. Uno de los puntos que queremos resaltar en esta sección es que nunca es demasiado pronto para comenzar las pruebas. Las pruebas y la experimentación tempranas presentan varias ventajas. En primer lugar, nos proporcionan una valiosa experiencia con un sistema, esto puede permitir detectar los problemas lo suficientemente pronto como para poder resolverlos, y con un coste mucho menor que si no los hubiéramos descubierto hasta mucho más adelante en el proceso de desarrollo. En segundo lugar, podemos empezar a construir una serie de casos y resultados de prueba que se pueden utilizar una y otra vez a medida que el sistema va creciendo. Cada vez que hagamos un cambio en el sistema, estos casos de prueba nos permitirán comprobar que no hemos introducido inadvertidamente errores en el resto del sistema, como resultado de los cambios.

Para ilustrar esta técnica de prueba dentro de BlueJ, utilizaremos el proyecto *online-shop*, que representa un primitivo intento de desarrollo de un software para una tienda de ventas en línea (como Amazon.com). Nuestro proyecto contiene solo una parte muy pequeña de esta aplicación; específicamente, la parte que trata con los comentarios de los clientes acerca de los productos que están a la venta.

Abra el proyecto *online-shop*. Actualmente, solo contiene dos clases: `SalesItem` y `Comment`. La funcionalidad que queremos para esta parte de la aplicación –que se concentra únicamente en la gestión de los comentarios de los clientes– es la siguiente:

- Los artículos a la venta pueden crearse con una descripción y un precio.
- Pueden añadirse y eliminarse comentarios de clientes relativos a los artículos que están a la venta.
- Los comentarios incluyen el texto del comentario, el nombre del autor y una puntuación. La puntuación está comprendida en el rango de 1 a 5 (ambos incluidos).
- Cada persona solo puede dejar un comentario. Los intentos subsiguientes de dejar un comentario por parte del mismo autor serán rechazados.
- La interfaz de usuario (no implementada todavía en este proyecto) incluirá una pregunta que plantea: "*Was this comment helpful to you?*" ("¿Le ha resultado de utilidad este

comentario?"), con botones Yes y No. Cuando un usuario haga clic sobre Yes o No estará *votando a favor* o *en contra* del comentario. Para cada comentario se almacena la diferencia entre votos a favor y en contra, de modo que podamos mostrar primero los comentarios más útiles (aquellos que tengan una mayor diferencia a favor).

La clase `Comment` de nuestro proyecto almacena información acerca de un único comentario. Para nuestras pruebas, nos centraremos en la clase `SalesItem`, mostrada en el Código 7.1. Los objetos de esta clase representan un único artículo de la tienda, incluyendo todos los comentarios que se han dejado en relación con este artículo.

Como parte de nuestras pruebas, debemos comprobar distintos aspectos de la funcionalidad deseada, incluyendo:

- ¿Pueden añadirse y eliminarse comentarios en un artículo de la tienda?
- ¿Muestra correctamente el método `showInfo` toda la información almacenada acerca de un artículo de la tienda?
- ¿Se están imponiendo correctamente las restricciones (la puntuación debe estar entre 1 y 5, solo se admite un comentario por persona)?
- ¿Podemos localizar correctamente el comentario más útil (el que tenga más votos)?

Veremos que todos estos aspectos pueden probarse cómodamente utilizando el banco de objetos de BlueJ. Además, veremos que la naturaleza interactiva de BlueJ permite simplificar partes de las pruebas, realizando modificaciones controladas en las clases que estemos probando.

Código 7.1
La clase `SalesItem`.

```
import java.util.ArrayList;
import java.util.Iterator;

/**
 * La clase representa los artículos que están a la venta en un sitio
 * de comercio electrónico en línea (como Amazon.com). Los objetos
 * SalesItem almacenan toda la información relativa a cada artículo,
 * incluyendo su descripción, su precio, los comentarios de los
 * clientes, etc.
 *
 * NOTA: ¡La versión actual está incompleta! Actualmente, solo se
 * incluye el código que gestiona los comentarios de los clientes.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 0.1 (2011-07-31)
 */
public class SalesItem
{
    private String name;
    private int price;  // en céntimos
    private ArrayList<Comment> comments;
```

Código 7.1
(continuación)
La clase `SalesItem`.

```
    /**
     * Crear un nuevo artículo.
     */
    public SalesItem(String name, int price)
    {
        this.name = name;
        this.price = price;
        comments = new ArrayList<Comment>();
    }

    /**
     * Devolver el nombre de este artículo.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Devolver el precio de este artículo.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Devolver el número de comentarios de clientes
     * para este artículo.
     */
    public int getNumberOfComments()
    {
        return comments.size();
    }

    /**
     * Añadir un comentario a la lista de comentarios de este
     * artículo. Devolver true si la operación tiene éxito
     * y false si se rechaza el comentario.
     *
     * El comentario se rechazará si el mismo autor ya ha dejado
     * un comentario o si la puntuación no es valida.
     * Las puntuaciones válidas son los números comprendidos
     * entre 1 y 5 (ambos incluidos).
     */
   public boolean addComment(String author, String text, int rating)
   {
        if(ratingInvalid(rating)) {  // rechaza las puntuaciones
                                     // no válidas
```

Código 7.1
(continuación)
La clase SalesItem.

```
            return false;
        }

        if(findCommentByAuthor(author) != null) {
            // rechaza varios comentarios del mismo autor
            return false;
        }
        comments.add(new Comment(author, text, rating));
        return true;
    }

    /**
     * Eliminar el comentario almacenado en el índice especificado.
     * Si el índice no es válido, no hacer nada.
     */
    public void removeComment(int index)
    {
        if(index >=0 && index < comments.size()) { // el índice es
                                                    // válido
            comments.remove(index);
        }
    }

    /**
     * Votar a favor del comentario correspondiente al índice
     * especificado. Es decir, considerar este comentario como
     * más útil. Si el índice no es válido, no hacer nada.
     */
    public void upvoteComment(int index)
    {
        if(index >=0 && index < comments.size()) { // el índice es
                                                    // válido.
            comments.get(index).upvote();
        }
    }

    /**
     * Votar en contra del comentario correspondiente al índice
     * especificado. Es decir, considerar este comentario como
     * menos útil. Si el índice no es válido, no hacer nada.
     */
    public void downvoteComment(int index)
    {
        if(index >=0 && index < comments.size()) { // el índice es
                                                    // válido.
            comments.get(index).downvote();
        }
    }
```

Código 7.1
(*continuación*)
La clase `SalesItem`.

```
    /**
     * Mostrar todos los comentarios en pantalla. (Actualmente, para
     * propósitos de prueba: imprimir en el terminal. Modificar
     * posteriormente para visualización web.)
     */
    public void showInfo()
    {
        System.out.println("*** " + name + " ***");
        System.out.println("Price: " + priceString(price));
        System.out.println();
        System.out.println("Customer comments:");
        for(Comment comment : comments) {
            System.out.println("-------------------------------------");
            System.out.println(comment.getFullDetails());
        }
        System.out.println();
        System.out.println("=====================================");
    }

    /**
     * Devolver el comentario más útil. El comentario más útil es
     * aquel que tenga mayor diferencia entre votos a favor y en
     * contra. Si hay varios comentarios empatados con la máxima
     * puntuación, devolver cualquiera de ellos.
     */
    public Comment findMostHelpfulComment()
    {
        Iterator<Comment> it = comments.iterator();
        Comment best = it.next();
        while(it.hasNext()) {
            Comment current = it.next();
            if(current.getVoteCount() > best.getVoteCount()) {
                best = current;
            }
        }
        return best;
    }

    /**
     * Comprobar si la puntuación dada no es válida. Devolver
     * true si no es válida. Las puntuaciones válidas están
     * en el rango [1..5].
     */
    private boolean ratingInvalid(int rating)
    {
        return rating < 0 || rating > 5;
    }
```

Código 7.1
(continuación)
La clase `SalesItem`.

```
    /**
     * Localizar el comentario hecho por el autor con el nombre
     * especificado.
     *
     * @return El comentario si existe o null si no existe.
     */
    private Comment findCommentByAuthor(String author)
    {
        for(Comment comment : comments) {
            if(comment.getAuthor().equals(author)) {
                return comment;
            }
        }
        return null;
    }

    /**
     * Para un precio dado como un int, devolver un objeto String
     * legible que represente el mismo precio. El precio está
     * expresado en centavos. Por ejemplo, para price==12345,
     * se devolverá el siguiente objeto String: $123.45
     */
    private String priceString(int price)
    {
        int dollars = price / 100;
        int cents = price - (dollars*100);
        if(cents <= 9) {
            return "$" + dollars + ".0" + cents;  // rellenar con
                                                  // ceros
        }
        else {
            return "$" + dollars + "." + cents;
        }
    }
}
```

7.3.1 Utilización de inspectores

A la hora de probar interactivamente, la utilización de inspectores de objetos suele ser muy útil. Como preparación para nuestras pruebas, cree un objeto `SalesItem` en el banco de objetos y abra su inspector seleccionando la función *Inspect* en el menú del objeto. Seleccione el campo `comments` y abra su inspector también (Figura 7.1). Compruebe que la lista se ha creado (no es `null`) y tiene inicialmente un tamaño igual a 0. Verifique también que el tamaño crece a medida que se añaden comentarios. Deje abierto el inspector de la lista de comentarios como ayuda para las siguientes pruebas.

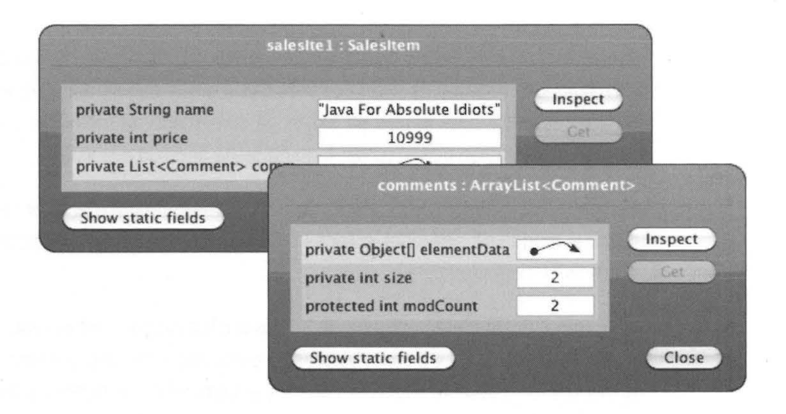


Figura 7.1
Inspector de la lista `comments`.

Un componente esencial a la hora de probar clases que utilicen estructuras de datos es comprobar que se comportan apropiadamente tanto cuando las estructuras de datos están vacías, como cuando están llenas –si es que ese caso puede darse. La prueba con estructuras de datos llenas solo se aplica a aquellas que tienen un límite fijo, como las matrices. En nuestro caso, donde estamos usando un `ArrayList`, no sería de aplicación para el caso de que la lista esté llena, porque la lista se irá expandiendo según sea necesario. Sin embargo, hacer pruebas con un lista vacía es importante, ya que este es un caso especial que necesita un tratamiento también especial.

Una primera prueba que puede hacerse con `SalesItem` consiste en llamar a su método `showInfo` antes de añadir ningún comentario. Esto debería mostrar correctamente la descripción y el precio del artículo, pero sin ningún comentario.

Una característica clave de unas buenas pruebas consiste en asegurarse de que se comprueben los *límites*, porque esos suelen ser los puntos en los que las cosas dejan de funcionar. Los límites asociados con la clase `SalesItem` son, por ejemplo, la lista de comentarios vacía. Los límites establecidos para la clase `Comment` incluyen la restricción de las puntuaciones al rango comprendido entre 1 y 5. Las puntuaciones situadas en los dos extremos de este rango son casos límite. Será importante comprobar no solo las puntuaciones situadas en mitad de este rango, sino también las puntuaciones máxima y mínima posibles.

Para realizar las pruebas según lo que acabamos de comentar, cree un objeto `SalesItem` en el banco de objetos y realice los siguientes ejercicios como pruebas iniciales de la funcionalidad relativa a los comentarios. Si lleva a cabo con cuidado estas pruebas, debería descubrir dos errores contenidos en nuestro código.

Ejercicio 7.1 Añada varios comentarios al artículo, mientras observa el inspector de la línea de comentarios. Asegúrese de que la lista se comporta de la forma esperada (es decir, su tamaño debe incrementarse). También puede ser conveniente inspeccionar el campo *elementData* del objeto ArrayList.

Ejercicio 7.2 Compruebe que el método `showInfo` imprime correctamente la información del artículo, incluidos los comentarios. Pruebe el método para artículos que tengan comentarios y artículos que no los tengan.

Ejercicio 7.3 Compruebe que el método `getNumberOfComments` funciona de la forma esperada.

Ejercicio 7.4 Ahora compruebe que se gestionan correctamente los autores duplicados; es decir, que se rechazan los comentarios ulteriores realizados por el mismo autor. Al tratar de añadir un comentario con un nombre de autor para el que ya exista un comentario, el método `addComment` debería devolver `false`. Compruebe también que el comentario no se ha añadido a la lista.

Ejercicio 7.5 Realice comprobaciones de los límites relativos al valor de puntuación. Es decir, cree comentarios que tengan no solo puntuaciones medias, sino también puntuaciones mínimas y máximas. ¿Funciona el mecanismo correctamente?

Ejercicio 7.6 Una buena prueba de límites implica probar también valores situados fuera del rango de datos válidos. Pruebe con 0 y 6 como valores de puntuación. En ambos casos, el comentario debería ser rechazado (`addComment` debería devolver `false` y el comentario no debería ser añadido a la lista).

Ejercicio 7.7 Pruebe los métodos `upvoteComment` y `downvoteComment`, que sirven para votar a favor y en contra de un comentario. Asegúrese de que el saldo de votos se mantiene correctamente.

Ejercicio 7.8 Utilice los métodos `upvoteComment` y `downvoteComment` para marcar algunos comentarios como más o menos útiles. Después, pruebe el método `findMostHelpfulComment`. Este método debería devolver el comentario que haya sido votado como más útil. Observará que el método devuelve una referencia a un objeto. Puede utilizar la función *Inspect* en el cuadro de diálogo de resultados del método para comprobar si se ha devuelto el comentario correcto. ¡Por supuesto, necesitará saber cuál es el comentario correcto para poder comprobar si ha obtenido el resultado adecuado!

Ejercicio 7.9 Realice una prueba de límites del método `findMostHelpfulComment`. Es decir, invoque este método cuando la lista de comentarios esté vacía (todavía no se han añadido comentarios). ¿Funciona el método de la forma esperada?

Ejercicio 7.10 Las pruebas de los ejercicios anteriores deberían haberle permitido descubrir dos errores en nuestro código. Corríjalos. Después de corregir esos errores, ¿se puede presuponer con seguridad que todas las pruebas anteriores podrán continuar funcionando como antes? En la Sección 7.4 se explican algunos de los problemas relativos a las pruebas que surgen cuando se corrige o mejora un software.

A partir de estos ejercicios, es fácil ver lo valiosos que son los inspectores y las invocaciones interactivas de métodos, a la hora de proporcionar información inmediata sobre el estado de un objeto, evitando a menudo la necesidad de añadir instrucciones de impresión a una clase para probarla o depurarla.

7.3.2 Pruebas positivas y negativas

Concepto:

Una **prueba positiva** es la prueba de un caso que se espera que funcione correctamente.

A la hora de decidir acerca de qué probar, distinguimos generalmente entre casos de prueba positivos y negativos. Una prueba positiva consiste en comprobar una funcionalidad que esperamos que sea correcta. Por ejemplo, si añadimos un comentario de un nuevo autor con una puntuación válida, estaremos haciendo una prueba positiva. Cuando trabajamos con casos de prueba positivos, lo que buscamos es convencernos de que el código ha funcionado como se esperaba.

Las pruebas negativas consisten en probar casos que esperamos que fallen. Utilizar una puntuación no válida o intentar almacenar un segundo comentario del mismo autor serían

Concepto:

Una **prueba negativa** es la prueba de un caso que se espera que falle.

casos de pruebas negativas. Al trabajar con pruebas negativas, esperamos que el programa maneje ese error de una forma especificada y controlada.

Error común Es un error muy común, en las personas poco experimentadas en las labores de pruebas, el realizar solo pruebas positivas. Las pruebas negativas –probar que lo que debe funcionar mal funciona efectivamente mal, y que lo hace de una manera bien definida– son cruciales para un buen procedimiento de pruebas.

Ejercicio 7.11 ¿Cuáles de los casos de prueba mencionados en los ejercicios anteriores son pruebas positivas y cuáles son negativas? Haga una tabla de cada categoría. ¿Puede pensar en otras pruebas positivas adicionales? ¿Puede pensar en otras pruebas negativas adicionales?

7.4 Automatización de pruebas

Una razón por la que no se suelen realizar unas pruebas exhaustivas es porque es una actividad que requiere mucho tiempo y es relativamente aburrida si se hace manualmente. Habrá observado, si realizó todos los ejercicios de la sección anterior, que probar exhaustivamente una aplicación puede llegar rápidamente a ser muy tedioso. Esto se convierte en un problema especialmente grave cuando las pruebas tienen que realizarse no solamente una vez, sino quizá muchos cientos o miles de veces. Afortunadamente, disponemos de técnicas que nos permiten automatizar las pruebas repetitivas, eliminando así buena parte del trabajo asociado. En la siguiente sección se examina la automatización de pruebas en el contexto de las *pruebas de regresión.*

7.4.1 Pruebas de regresión

Sería maravilloso si pudiéramos dar por supuesto que el corregir los errores siempre mejora la calidad de los programas. Lamentablemente, la experiencia demuestra que suelen introducirse con mucha facilidad errores adicionales al modificar el software. Así, corregir un error en un determinado lugar puede introducir otro error al mismo tiempo.

En consecuencia, es deseable ejecutar *pruebas de regresión* cada vez que se realiza un cambio en el software. Las pruebas de regresión implican volver a ejecutar pruebas que ya se pasaron anteriormente con éxito, para garantizar que la nueva versión sigue superándolas. Es mucho más probable que estas pruebas se lleven a cabo si las podemos automatizar. Una de las formas más sencillas de automatizar las pruebas de regresión consiste en escribir un programa que actúe como *marco de pruebas.*

7.4.2 Pruebas automatizadas mediante JUnit

BlueJ (y muchos otros entornos de desarrollo) incluye el soporte para las pruebas de regresión, utilizando un sistema de pruebas denominado JUnit. JUnit es un entorno de pruebas desarrollado por Erich Gamma y Kent Beck para el lenguaje Java y hoy día hay disponibles sistemas similares para muchos otros lenguajes de programación.

JUnit, *www.junit.org* JUnit es un entorno de pruebas popular que soporta la prueba estructurada de unidades y las pruebas de regresión en Java. Está disponible en una versión independiente del entorno de desarrollo específico que se esté utilizando, y también está disponible de forma integrada en muchos entornos. JUnit fue desarrollado por Erich Gamma y Kent Beck. Puede encontrar el software y una gran cantidad de información acera de él en `http://www.junit.org`.

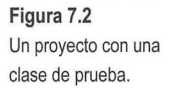
Para comenzar a investigar las pruebas de regresión con nuestro ejemplo, abra el proyecto *online-shop-junit*. Este proyecto contiene las mismas clases que el anterior más otra clase adicional, `SalesItemTest`.

`SalesItemTest` es una clase de prueba. Lo primero que hay que observar es que su apariencia es distinta de las que hemos visto anteriormente (Figura 7.2). Está anotada como `<<unit test>>`, su color es diferente del de las clases normales del diagrama y está asociada a la clase `SalesItem` (se desplazará con esta clase si movemos `SalesItem` en el diagrama).

Como puede ver, la Figura 7.2 también muestra algunos controles adicionales en la ventana principal, debajo del botón *Compile*. Esos controles permiten utilizar las herramientas integradas para la realización de pruebas de regresión. Si no ha utilizado anteriormente JUnit en BlueJ, las herramientas de pruebas estarán desactivadas y los botones no estarán visibles en el sistema. Lo que debe hacer ahora es activar esas herramientas de prueba. Para ello, acceda a la pestaña *Miscellaneous* del cuadro de diálogo *Preferences* (Preferencias) y asegúrese de que esté seleccionada la opción *Show unit testing tools* (Mostrar herramientas para la realización de pruebas de unidades).

Hay otra diferencia adicional que resulta aparente en el menú que se presenta en pantalla cuando hacemos clic con el botón derecho del ratón en la clase de prueba (Figura 7.3). En el menú hay tres nuevas secciones en lugar de una lista de constructores.

Utilizando las clases de prueba podemos automatizar las pruebas de regresión. La clase de prueba contiene el código para llevar a cabo una serie de pruebas preparadas y comprobar sus resultados. Esto hace que repetir las mismas pruebas sea mucho más sencillo.


Figura 7.2
Un proyecto con una clase de prueba.

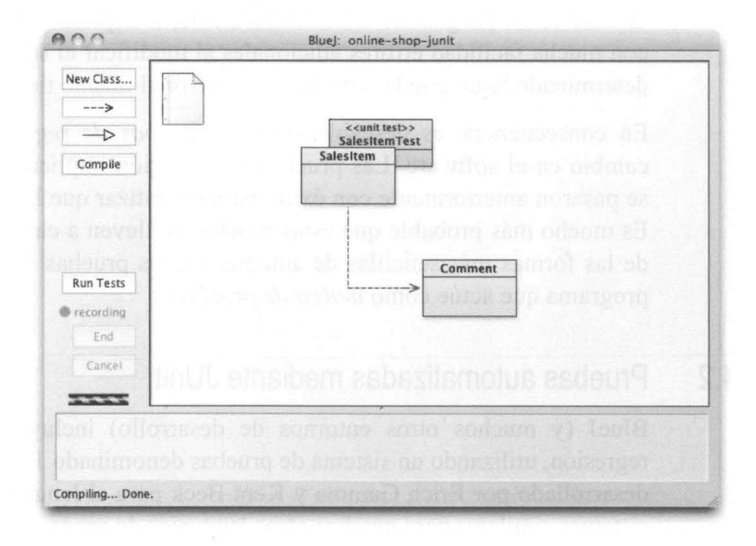




Figura 7.3
El menú emergente para una clase de prueba.

Normalmente, cada clase de prueba está asociada con una clase normal del proyecto. En este caso, `SalesItemTest` está asociada con la clase `SalesItem` y decimos que `SalesItem` es la *clase de referencia* de `SalesItemTest`.

En nuestro proyecto, la clase de prueba ya ha sido creada y ya contiene algunas pruebas. Ahora podemos ejecutar dichas pruebas haciendo clic en el botón *Run Tests* (Ejecutar pruebas).

Ejercicio 7.12 Ejecute las pruebas en su proyecto utilizando el botón *Run Tests*. Debería ver una ventana similar a la de la Figura 7.4, resumiendo los resultados de las pruebas.

Figura 7.4
La ventana que muestra los resultados de las pruebas.

La Figura 7.4 muestra el resultado de ejecutar tres pruebas denominadas `testAddComment`, `testIllegalRating` y `testInit`, que están definidas en la clase de prueba. Las marcas situadas inmediatamente a la izquierda de los nombres de cada prueba indican que las pruebas se han ejecutado con éxito. Puede obtener los mismos resultados seleccionando la opción *Test All* del menú emergente asociado con la clase de prueba o ejecutando las pruebas individualmente seleccionándolas una a una en el menú.

Las clases de prueba son claramente diferentes en cierto sentido de las clases normales, y si abre el código fuente de `SalesItemTest`, observará que tiene algunas nuevas características.

En esta etapa del libro no vamos a explicar en detalle cómo funcionan internamente las clases de prueba, pero merece la pena resaltar que aunque el código fuente de `SalesItemTest` podría haber sido escrito por una persona, de hecho ha sido *generado automáticamente* por BlueJ. Algunos de los comentarios se añadieron posteriormente para documentar el propósito de las pruebas.

Cada clase de prueba suele contener pruebas para verificar la funcionalidad de su clase de referencia. Se crea utilizando el botón derecho del ratón sobre una clase de referencia potencial y seleccionando *Create Test Class* (Crear clase de prueba) en el menú emergente. Observe que `SalesItem` ya tiene una clase de prueba, por lo que este elemento adicional del menú no aparecerá en su menú de clase, pero la clase `Comment` sí que dispone de esta opción, ya que actualmente no tiene ninguna clase de prueba asociada.

La clase de prueba contiene tanto código fuente para ejecutar pruebas sobre una clase de referencia, como para comprobar si las pruebas han tenido éxito o no. Por ejemplo, he aquí una de las instrucciones de `testInit` que comprueba que el precio del artículo sea 1000 en dicho punto:

```
assertEquals(1000, salesItel.getPrice());
```

Cuando se ejecutan pruebas así, BlueJ es capaz de presentar los resultados en la ventana que se muestra en la Figura 7.4.

En la siguiente sección, vamos a explicar cómo soporta BlueJ la creación de pruebas, para que podamos crear nuestras propias pruebas automatizadas.

Ejercicio 7.13 Cree una clase de prueba para la clase `Comment` en el proyecto *online-shop-junit*.

Ejercicio 7.14 ¿Qué métodos se crean automáticamente al crear una nueva clase de prueba?

7.4.3 Grabación de una prueba

Como hemos dicho al principio de la Sección 7.4, la automatización de pruebas es conveniente porque crear y volver a crear pruebas de forma manual es un proceso que lleva mucho tiempo. BlueJ permite combinar la efectividad de la prueba manual de unidades con la potencia de la automatización de pruebas al permitirnos grabar pruebas manuales y luego reproducirlas posteriormente con el objetivo de llevar a cabo pruebas de regresión. La clase `SalesItemTest` se creó mediante este proceso.

Suponga que quisiéramos probar exhaustivamente el método `addComment` de la clase `SalesItem`. Este método, como hemos visto, añade comentarios de los clientes siempre y cuando sean válidos. Hay varias pruebas que nos gustaría realizar, como por ejemplo:

- Añadir un primer comentario a una lista de comentarios vacía (positiva).
- Añadir comentarios adicionales cuando ya existen otros comentarios (positiva).
- Intentar añadir un comentario con un autor que ya ha enviado un comentario (negativa).
- Intentar añadir un comentario con un puntuación no válida (negativa).

La primera de estas pruebas ya existe en la clase `SalesItemTest`. Ahora describiremos cómo crear la siguiente utilizando el proyecto *online-shop-junit*.

Las pruebas se graban indicándole a BlueJ que comience a grabar, realizando la prueba manualmente y luego indicando que la prueba ha finalizado. El primer paso se lleva a cabo a través del menú asociado a una clase de prueba. Esto le dice a BlueJ en qué clase queremos almacenar la nueva prueba. Seleccione *Create Test Method...* (Crear método de prueba...) en el menú emergente de la clase `SalesItemTest`. Se le pedirá que especifique un nombre para el método de prueba. Por convenio, comenzamos los nombres de las pruebas con el prefijo *test*. Por ejemplo, para crear un método que añada dos comentarios, podríamos llamar a dicho método `testTwoComments`.[1]

Una vez que haya introducido un nombre y haya hecho clic en OK, aparecerá un indicador de grabación de color rojo a la izquierda del diagrama de clases y los botones *End* (Finalizar) y *Cancel* (Cancelar) pasarán a estar disponibles. *End* se usa para indicar el final del proceso de creación de la prueba y *Cancel* para abandonarlo.

Una vez iniciada la grabación, nos limitaremos a llevar a cabo las acciones que ejecutaríamos con un prueba manual normal:

- Crear un objeto `SalesItem`.
- Añadir un comentario al artículo.

Concepto:

Una **aserción** es una expresión que establece una condición que esperamos que sea cierta. Si la condición es falsa, decimos que la aserción ha fallado. Esto indica que hay algún error en nuestro programa.

Una vez invocado `addComment`, aparecerá un nuevo cuadro de diálogo (Figura 7.5). Se trata de una versión ampliada de la ventana de resultados de método normal, y es una parte crucial del proceso de automatización de las pruebas. Su propósito es permitirnos especificar cuál *debería ser* el resultado de la llamada al método. Esto recibe el nombre de *aserción*.

En este caso, esperamos que el valor de retorno del método sea *true*, y queremos incluir una comprobación en nuestra prueba para asegurarnos de que efectivamente es así. Por tanto, deberemos cerciorarnos de que la casilla de verificación *Assert that* (incluir aserción) esté marcada, introducir *true* en el campo y seleccionar el botón *Close*.

- Añada un segundo comentario a su artículo. Asegúrese de que los comentarios sean válidos (que los autores sean distintos y que la puntuación sea válida). Defina la aserción de que el resultado sea también cierto para el segundo comentario.

BlueJ: Method Result
// Add a comment to the comment list of this sales item. Return true if successful;
// false if the comment was rejected.
//
// The comment will be rejected if the same author has already left a comment, or
// if the rating is invalid. Valid ratings are numbers between 1 and 5 (inclusive).
boolean addComment(String author, String text, int rating)
salesIte1.addComment("James Duckling", "Great book. I felt like I understood Java for the first time.", 4)
returned:
boolean true
Inspect
Get
Assert that:
result is equal to
Close

Figura 7.5
Cuadro de diálogo *Method Result* que indica el resultado del método, con la funcionalidad de aserción añadida.

[1] Las versiones anteriores de JUnit, hasta la versión 3, exigían que los nombres de los métodos comenzaran con el prefijo *test*. En las versiones actuales, esto ya no es un requisito

- Ahora esperamos que existan dos comentarios. Para comprobar que efectivamente esto es así, invoque el método `getNumberOfComments` y establezca la aserción de que el resultado tiene que ser 2.

Esta es la etapa final de la prueba. A continuación pulsaremos el botón *End* para detener la grabación. En este punto, BlueJ añadirá el código fuente a la clase `SalesItemTest` para nuestro nuevo método, `testTwoComments`, a continuación compilará la clase y limpiará el banco de objetos. El método generado resultante se muestra en el Código 7.2.

Código 7.2
Un método de prueba generado automáticamente.

```
@Test
public void testTwoComments()
{
    SalesItem salesIte1 = new SalesItem("Java Book", 12345);
    assertEquals(true, salesIte1.addComment("James Duckling",
                                "Great book. . .", 4));
    assertEquals(true, salesIte1.addComment("Fred", "Like it", 2));
    assertEquals(2, salesIte1.getNumberOfComments());
}
```

Como puede ver, el método contiene instrucciones que reproducen las acciones que hemos llevado a cabo al grabarlo: se crea un objeto `SalesItem` y se invocan los métodos `addComment` y `getNumberOfComments`. La llamada a `assertEquals` es la que se encarga de comprobar que el resultado devuelto por estos métodos se corresponde con el valor esperado. También puede ver una nueva estructura, `@Test`, antes del método. Es una anotación que identifica este método como un método de prueba.

Los siguientes ejercicios se proporcionan para que pueda probar este proceso por sí mismo. Incluyen un ejemplo que muestra lo que sucede si el valor obtenido no se corresponde con el valor esperado.

Ejercicio 7.15 Cree una prueba para comprobar que `addComment` devuelve `false` cuando ya existe otro comentario del mismo autor.

Ejercicio 7.16 Cree una prueba que realice pruebas negativas de los límites del rango de puntuación. Es decir, pruebe los valores 0 y 6 como puntuación (los valores justo fuera del rango legal). Esperamos que estos valores hagan que se devuelva `false`, así que establezca en el cuadro de diálogo de resultados la aserción de que el resultado ha de ser `false`. Observará que con uno de estos valores se devuelve actualmente `true` (de manera incorrecta). Este es el error que ya descubrimos anteriormente al realizar las prueba manuales. Asegúrese de todos modos de establecer `false` como aserción. La aserción establece el resultado *esperado*, no el resultado *real*.

Ejercicio 7.17 Ejecute de nuevo todas las pruebas. Observe cómo muestra el cuadro de diálogo *Test Result* las pruebas que han fallado. Seleccione en la lista la prueba fallida. ¿Qué opciones tiene disponibles para explorar los detalles de la prueba fallida?

Ejercicio 7.18 Cree una clase de prueba que tenga `Comment` como clase de referencia. Cree una prueba que verifique si los detalles relativos al autor y a la puntuación se han almacenado correctamente después de la creación del comentario. Grabe pruebas separadas que verifiquen si los métodos `upvote` y `downvote` funcionan de la forma esperada.

Ejercicio 7.19 Cree pruebas para `SalesItem` que verifiquen si el método `findMostHelpfulComment` funciona de la forma esperada. Observe que este objeto devuelve un objeto `Comment`. Durante la prueba puede utilizar el botón *Get* en el cuadro de diálogo de resultados para cargar el objeto resultado en el banco de objetos, lo que le permitirá realizar otras llamadas a métodos y añadir aserciones para este objeto. Con ello, podrá identificar el objeto comentario devuelto (por ejemplo, comprobando quién es el autor). También puede establecer como aserción que el resultado sea *null* o *not null*, dependiendo de lo que espere obtener.

7.4.4 Bancos de pruebas

Concepto:

Un **banco de pruebas** es un conjunto de objetos en un estado definido que sirven como base para realizar pruebas de unidades.

A medida que desarrollamos un conjunto de métodos de prueba es habitual encontrarse con que tenemos que crear objetos similares para cada uno de ellos. Por ejemplo, toda prueba de la clase `SalesItem` implicará crear al menos un objeto `SalesItem` e inicializarlo, a menudo añadiendo uno o más comentarios. Un objeto o grupo de objetos que se emplean en más de una prueba se denomina *banco de pruebas (fixture)*. Hay dos elementos de menú asociados con las clases de prueba que nos permiten trabajar con los bancos de prueba en BlueJ: *Object Bench to Test Fixture* (Del banco de objetos al banco de pruebas) y *Test Fixture to Object Bench* (Del banco de pruebas al banco de objetos). En su proyecto, cree dos objetos `SalesItem` en el banco de objetos. Deje uno de ellos sin comentarios y añada dos comentarios al otro. Ahora seleccione *Object Bench to Test Fixture* en `SalesItemTest`. Los objetos desaparecerán del banco de objetos y si examina el código fuente de `SalesItemTest`, verá que su método `setUp` tiene un aspecto similar al Código 7.3, donde `salesIte1` y `salesIte2` se han definido como campos.

Código 7.3
Creación de un banco de pruebas.

```
/**
 * Define el banco de pruebas.
 *
 * Se invoca antes del método correspondiente a cada caso de prueba.
 */
@Before
public void setUp()
{
    salesIte1 = new SalesItem("Java Book", 12345);
    salesIte2 = new SalesItem("Other", 123);
    salesIte2.addComment("Fred", "too expensive", 1);
}
```

La importancia del método `setUp` es que se invoca de forma automática inmediatamente antes de invocar cada método de prueba. (La anotación `@Before` situada delante de la cabecera del método se encarga de garantizar esto.) Esto implica que los métodos de prueba individuales ya no necesitan crear sus propias versiones de los objetos contenidos en el banco de pruebas.

Una vez que hemos asociado un banco de pruebas con una clase de prueba, grabar las pruebas resulta bastante más sencillo. Cada vez que creemos un nuevo método de prueba, los objetos del banco de pruebas aparecerán automáticamente en el banco de objetos –ya no habrá necesidad de crear nuevos objetos de prueba manualmente cada vez.

Si quisiéramos añadir más objetos al banco de pruebas en cualquier momento, una de las formas más fáciles consiste en seleccionar *Test Fixture to Object Bench*, añadir objetos adicionales al banco de objetos de la manera usual y luego seleccionar *Object Bench to Test Fixture*. También podríamos editar el método `setUp` en el editor y añadir campos adicionales directamente en la clase de prueba.

La automatización de las pruebas es un concepto muy potente porque aumenta la probabilidad de que el programador llegue a escribir las pruebas y también hace más probable que esas pruebas se ejecuten una y otra vez a medida que se desarrolla el programa. Debería tratar de formarse el hábito de escribir pruebas de unidades en una etapa temprana del desarrollo de un proyecto, y mantenerlas actualizadas a medida que el proyecto progresa. En el Capítulo 12, volveremos a hablar del tema de las aserciones en el contexto del tratamiento de errores.

Ejercicio 7.20 Añada pruebas automatizadas adicionales a su proyecto, hasta que alcance un punto en el que esté razonablemente confiado en la correcta operación de las clases. Utilice pruebas positivas y negativas. Si descubre cualquier error, asegúrese de grabar pruebas que protejan contra la reaparición de dichos errores en versiones posteriores.

En la siguiente sección vamos a examinar el tema de la depuración, la actividad que comienza cuando hemos observado la existencia de un error y necesitamos localizarlo y corregirlo.

7.5 Depuración

Las pruebas son importantes y ayudan a descubrir los errores existentes. Sin embargo, las pruebas por sí mismas no son suficiente. Después de detectar la existencia de un error, tenemos que encontrar también su causa y corregirlo. Ahí es donde entra en acción el proceso de depuración.

Para explicar las distintas técnicas de depuración vamos a utilizar un escenario hipotético. Imagine que le han pedido que se una a un equipo de proyecto existente que está trabajando en la implementación de una calculadora software (Figura 7.6). Se le ha elegido porque uno de los principales miembros del equipo de programación, Hacker T. Largebrain, acaba de ser ascendido a un puesto directivo en otro proyecto. Antes de partir, Hacker aseguró al equipo al que le acaban de asignar que su implementación de la parte de la calculadora de la que era responsable estaba finalizada y completamente probada. Había escrito incluso algún software de prueba para verificar que efectivamente era así. Lo que le piden a usted es que se haga cargo de la clase y simplemente verifique que está adecuadamente comentada antes de integrarla con las clases que están escribiendo otros miembros del equipo.

El software para la calculadora ha sido diseñado cuidadosamente para separar la interfaz de usuario de la lógica de la calculadora, de modo que la calculadora pueda utilizarse en diferentes contextos más adelante. La primera versión, que es la que estamos examinando aquí, se ejecutará con una interfaz gráfica de usuario, mostrada en la Figura 7.6. Sin embargo, en posteriores ampliaciones del proyecto, se pretende que la misma implementación de la calculadora pueda ejecutarse en un explorador web o en un dispositivo móvil. Para prepararse para esto, la aplicación se ha dividido en clases separadas, siendo las dos más importantes `UserInterface`, que implementa la interfaz gráfica de usuario y `CalcEngine`, que implementa la lógica de cálculo. Es esta última clase la que era responsabilidad de Hacker. Esta clase no

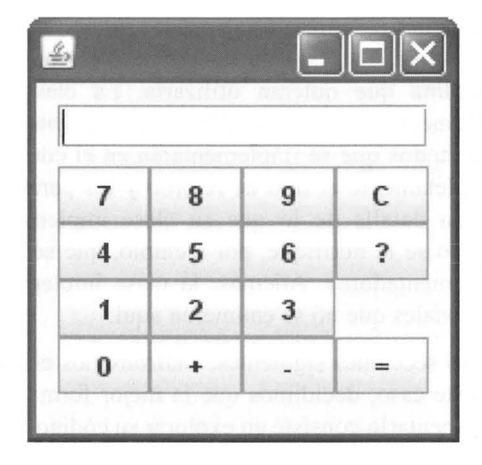


Figura 7.6
La interfaz de usuario de una calculadora software.

debe requerir ninguna modificación cuando la calculadora se ejecute con una interfaz de usuario diferente.

Para investigar cómo es utilizada la clase `CalcEngine` por otras clases, nos será útil examinar su *interfaz*. Aunque sea algo confuso, ahora no estamos hablando de la *interfaz de usuario*, sino de la *interfaz de la clase*. Este doble significado del término *interfaz* es un engorro, pero es importante comprender ambos significados.

La interfaz de la clase es el resumen de las cabeceras de los métodos públicos que otras clases pueden utilizar. Eso es lo que otras clases podrán ver y también define la forma en que esas clases podrán interactuar con la nuestra. La interfaz se muestra en la documentación `javadoc` de una clase y en la vista *Documentation* del editor. El Código 7.4 muestra la interfaz de la clase `CalcEngine`.

Código 7.4
La interfaz de la unidad aritmético-lógica.

```
// Devuelve el valor que hay que visualizar.
public int getDisplayValue();

// Invocación cuando se pulsa el botón de un dígito.
public void numberPressed(int number);

// Invocación cuando se pulsa el operador de suma.
public void plus();

// Invocación cuando se pulsa el operador de resta.
public void minus();

// Invocación para completar un cálculo.
public void equals();

// Invocación para reinicializar la calculadora.
public void clear();
```

Una interfaz como esta puede escribirse antes de implementar las clases completas. Representa una especie de contrato simple entre la clase `CalcEngine` y otras partes del programa que quieran utilizarla. La clase `CalcEngine` se encarga de proporcionar la implementación correspondiente a esta interfaz. La interfaz describe un conjunto mínimo de métodos que se implementarán en el componente lógico, y para cada método se definen completamente el tipo de retorno y los parámetros. Observe que la interfaz no proporciona ningún detalle de lo que su clase implementadora hará exactamente de manera interna cuando se la notifique, por ejemplo, que se ha pulsado el operador suma, eso se deja a los implementadores. Además, la clase implementadora puede disponer también de métodos adicionales que no se enumeren aquí.

En las secciones siguientes, analizaremos el intento de Hacker de implementar esta interfaz. En este caso, decidimos que la mejor forma de comprender el software de Hacker antes de documentarlo consiste en explorar su código fuente y el comportamiento de sus objetos.

7.6 Comentarios y estilo

Abra el proyecto *calculator-engine* para ver las clases. Esta es la versión de Hacker del proyecto que contiene solo el motor de la calculadora y una clase de prueba, pero no la clase correspondiente a la interfaz de usuario. La clase `CalcEngineTester` asume el papel de la interfaz de usuario en esta etapa del desarrollo. Esto ilustra otra ventaja de definir las interfaces entre las clases: es más sencillo desarrollar simuladores de las otras clases con propósitos de prueba.

Si echa un vistazo a la clase `CalcEngine`, comprobará que su autor ha prestado atención a varios aspectos relacionados con el buen estilo de programación:

- La clase tiene un comentario de varias líneas al principio, que indica el propósito de la clase. También incluye anotaciones que indican el autor y el número de versión.
- Cada método de la interfaz tiene un comentario que indica su propósito, sus parámetros y el tipo de retorno. Esto facilitará, ciertamente, el generar la documentación del proyecto para la interfaz, como se explica en el Capítulo 5.
- La disposición de las clases es coherente, con una cantidad apropiada de espacios en blanco, sangrado, que se utilizan para indicar los distintos niveles de bloques anidados y estructuras de control.
- Se han elegido nombres de variables y de métodos suficientemente expresivos.

Aunque estos convenios pueden parecer una pérdida de tiempo durante la implementación, pueden representar una enorme ventaja a la hora de ayudar a alguna otra persona a entender nuestro código (como hemos visto en este escenario), o a la hora de ayudarnos a nosotros mismos a recordar lo que hacía una clase, si por cualquier motivo interrumpimos nuestro trabajo en la misma.

También podemos observar en el código otro detalle que parece menos prometedor: Hacker no ha utilizado una clase de prueba de unidades especializada para capturar sus pruebas, sino que ha escrito su propia clase de prueba. Como sabemos perfectamente que BlueJ soporta las pruebas de unidades, nos preguntamos por qué Hacker habrá hecho eso.

Esto no necesariamente tiene que ser malo. Las clases de prueba escritas de forma manual pueden ser igual de buenas, aunque nos hace sospechar un poco. ¿Sabía Hacker realmente lo que estaba haciendo? Volveremos sobre este punto un poco más adelante.

¿Es posible que las habilidades de Hacker sean tan grandes como el piensa que son y que en ese caso no tengamos que dedicar mucho esfuerzo a hacer que la clase esté lista para integrarla con las otras? Trate de hacer los siguientes ejercicios para ver si es así.

Ejercicio 7.21 Asegúrese de que las clases del proyecto están compiladas y luego cree en BlueJ un objeto `CalcEngineTester`. Invoque el método `testAll`. ¿Qué es lo que se muestra en la ventana del terminal? ¿Realmente se cree la línea final del texto que aparece?

Ejercicio 7.22 Utilizando el objeto que ha creado en el ejercicio anterior, invoque el método `testPlus`. ¿Qué resultado proporciona? ¿Es el mismo resultado que se obtuvo al invocar `testAll`? Invoque `testPlus` una vez más. ¿Qué resultado obtiene ahora? ¿Debería dar siempre la misma respuesta? En caso afirmativo, ¿cuál tendría que ser esa respuesta? Examine el código fuente del método `testPlus` para comprobarlo.

Ejercicio 7.23 Repita el ejercicio anterior con el método `testMinus`. ¿Proporciona siempre el mismo resultado?

Los experimentos anteriores deberían alertarnos de que la clase `CalcEngine` no es completamente correcta. Parece que contiene algunos errores. ¿Pero cuáles son esos errores y cómo podemos encontrarlos? En las siguientes secciones, veremos una serie de formas distintas en las que podemos tratar de localizar dónde se encuentran los errores dentro de una clase.

7.7 Recorridos manuales

Concepto:

Un **recorrido manual** es la actividad consistente en analizar un segmento de código línea a línea mientras que se observan los cambios de estado y otros comportamientos de la aplicación.

Los recorridos manuales son un técnica relativamente poco utilizada, quizá porque es una técnica de depuración y pruebas particularmente "poco sofisticada". Sin embargo, no permita que esto le induzca a pensar que no son una herramientas útil. Un recorrido manual implica imprimir copias de las clases que se está intentando comprender o depurar y trabajar alejado de la computadora. Es demasiado fácil invertir un montón de tiempo sentado delante de la pantalla de la computadora, sin hacer demasiados progresos al tratar de resolver un problema de programación. Sentarse en algún otro lugar y tratar de centrar en una nueva dirección nuestros esfuerzos puede a menudo liberar nuestra mente, para atacar un problema desde una perspectiva completamente distinta. Según nuestra experiencia, parar para tomar el almuerzo o para hacer algo de deporte suele ayudar a arrojar luz sobre un problema que nos ha estado eludiendo durante horas interminables de trabajo delante del teclado.

Un recorrido manual implica leer las clases y trazar el flujo de control entre las clases y objetos. Esto ayuda a comprender tanto la forma en que los objetos interactúan entre sí, como la manera en que se comportan internamente. De hecho, un recorrido manual es una simulación hecha con lápiz y papel de lo que sucede dentro de la computadora cuando se ejecuta un programa. En la práctica, lo mejor es centrarse en una pequeña parte de la aplicación, como un único agrupamiento lógico de acciones o incluso una única llamada a método.

7.7.1 Un recorrido de alto nivel

Ilustraremos la técnica del recorrido manual con el proyecto *calculator-engine*. Es posible que sea conveniente que imprima copias de las clases `CalcEngine` y `CalcEngineTester` para poder ir siguiendo los pasos del proceso.

Comenzaremos examinando el método `testPlus` de la clase `CalcEngineTester`, ya que contiene una única agrupación lógica de acciones que nos debería ayudar a entender cómo funcionan conjuntamente varios métodos de la clase `CalcEngine` para llevar a cabo las tareas de computación propias de una calculadora. A medida que realicemos el recorrido, tomaremos nota de las cuestiones que se nos plantean.

1. Para esta primera etapa, no queremos perdernos en excesivos detalles. Simplemente queremos ver cómo el método `testPlus` utiliza el objeto encargado del motor (`engine`) de la calculadora, sin explorar los detalles internos de este. A partir de nuestros experimentos anteriores, parece que es preciso encontrar algunos errores, pero no sabemos si esos errores se encuentran en el método de prueba o en el motor. Por tanto, el primer paso consiste en comprobar que el método de prueba está utilizando el motor apropiadamente.

2. Observamos que la primera instrucción de `testPlus` supone que el campo `engine` ya hace referencia a un objeto válido:

   ```
   engine.clear();
   ```

   Podemos verificar que esto es así comprobando el constructor de la clase de prueba. Uno de los errores más comunes consiste en no inicializar correctamente los campos de un objetos, bien en sus declaraciones o en un constructor. Si tratamos de utilizar un campo que no tenga ningún objeto asociado, entonces es probable que se produzca un error `NullPointerException` en tiempo de ejecución.

3. La llamada a `clear` en la primera instrucción parece ser un intento de poner el motor de la calculadora en un estado inicial válido, listo para recibir instrucciones con el fin de realizar algún cálculo. Esto parece algo bastante razonable, equivalente a pulsar la tecla "reset" o "clear" en una calculadora real. En esta etapa, no examinamos la clase `engine` para comprobar qué es lo que hace exactamente el método `clear`. Eso puede esperar hasta que tengamos una razonable confianza en que las acciones de la clase de prueba son razonables. En lugar de ello, nos limitamos a anotar que hay que comprobar más adelante si `clear` pone el motor en un estado inicial válido, como se espera que haga.

4. La siguiente instrucción de `testPlus` es la introducción de un dígito mediante el método `numberPressed`:

   ```
   engine.numberPressed(3);
   ```

   Esto también es razonable, ya que el primer paso a la hora de realizar un cálculo consiste en introducir el primer operando. Una vez más, no examinamos qué es lo que hace el motor con ese número. Simplemente asumimos que lo almacena en alguna parte para usarlo posteriormente en el cálculo.

5. La siguiente instrucción llama a `plus`, por lo que deducimos que el valor completo del operando izquierdo es 3. Podemos tomar nota de este hecho en la versión impresa de la clase de prueba o marcar con una señal esta aserción en uno de los comentarios de `testPlus`. Igualmente, deberíamos confirmar ahora o más adelante que la operación que se está ejecutando es en efecto una suma. Esto parece bastante trivial, pero no es nada

infrecuente que los comentarios de una clase estén equivocados con respecto al código que se supone que documentan. Por tanto, verificar los comentarios al mismo tiempo que leemos el código puede ayudarnos a evitar que esos comentarios nos induzcan a error más adelante.

6. A continuación, se introduce otro dígito como operando derecho mediante una llamada adicional a `numberPressed`.

7. La finalización de la suma se solicita mediante una llamada al método `equals`. Podemos tomar nota por escrito de que, teniendo en cuenta la forma en que se ha usado en `testPlus`, el método `equals` no parece devolver el resultado del cálculo, como hubiéramos podido imaginar. Esta es otra cosa más que podemos comprobar cuando examinemos `CalcEngine`.

8. La última instrucción de `testPlus` obtiene el valor que se mostrará en la pantalla de la calculadora:

   ```
   return engine.getDisplayValue();
   ```

   Presumiblemente, este es el resultado de la suma, pero no podemos estar seguros sin examinar `CalcEngine` en detalle. De nuevo, tomaremos nota para comprobar más adelante que efectivamente es así.

Habiendo completado nuestro examen de `testPlus`, tenemos una razonable confianza en que el método de prueba está utilizando el motor adecuadamente: es decir, que está simulando una secuencia reconocible de pulsaciones de tecla para llevar a cabo un cálculo simple. Podríamos tomar nota del hecho de que el método no es particularmente ambicioso –ambos operandos son números de un único dígito y solo se emplea un único operador. Sin embargo, esto no es infrecuente en los métodos de prueba, porque es importante comprobar la funcionalidad más básica antes de probar combinaciones más complejas. De todos modos, es útil observar que habría que añadir algunas pruebas más complejas a la clase de pruebas en algún momento.

Ejercicio 7.24 Realice un recorrido similar del método `testMinus`. ¿Le sugiere esto alguna cuestión adicional sobre cosas que habría que comprobar al examinar en detalle `CalcEngine`?

Antes de examinar la clase `CalcEngine`, merece la pena hacer un recorrido del método `testAll` para ver cómo utiliza los métodos `testPlus` y `testMinus` que hemos estado examinando. Al hacerlo, observamos lo siguiente:

1. El método `testAll` es una secuencia lineal de instrucciones de impresión.

2. Contiene una llamada a `testPlus` y otra a `testMinus`, imprimiéndose los valores que esos métodos devuelven para que los usuarios los vean. Podemos observar que no hay nada que indique al usuario cuáles deberían ser esos valores. Esto hace que al usuario le resulte difícil confirmar que los resultados son correctos.

3. La línea final de texto se limita a afirmar que todas las pruebas se han pasado con éxito:

   ```
   All tests passed.
   ```

   pero el método no contiene ninguna prueba para establecer la veracidad de esta afirmación. Debería haber alguna forma de establecer cuáles tendrían que ser los resultados y si se han calculado correctamente o no. Esto es algo que deberíamos remediar en cuanto tengamos la oportunidad de ponernos a trabajar con el código fuente de esta clase.

En esta etapa, no deberíamos dejarnos distraer por este último punto y tenemos que resistirnos a la tentación de ponernos a hacer cambios que no pretendan directamente corregir los errores que estamos buscando. Si nos pusiéramos a hacer otros cambios, podríamos terminar muy fácilmente enmascarando los errores. Uno de los requisitos fundamentales para poder realizar adecuadamente las tareas de depuración consiste en ser capaz de reproducir el error que estamos buscando de una manera fácil y predecible. Cuando eso se consigue, es mucho más sencillo comprobar el efecto de nuestros intentos de corrección.

Habiendo comprobado la clase de prueba, ya estamos en disposición de examinar el código fuente de la clase `CalcEngine`. Podemos hacerlo armados con una secuencia razonable de llamadas a métodos que tenemos que explorar, secuencia que hemos extraído gracias a nuestro recorrido manual del método `testPlus`; también disponemos, asimismo, de una serie de cuestiones que se nos plantearon al hacer ese recorrido manual.

7.7.2 Comprobación del estado mediante un recorrido

Un objeto `CalcEngine` tiene un estilo muy distinto al del objeto que se utiliza para comprobarlo. Esto se debe a que el motor de la calculadora es un objeto completamente pasivo: no inicia por sí mismo ninguna actividad, sino que se limita a responder a llamadas a método externas. Esto es típico del estilo de comportamiento propio de los servidores. Los objetos servidores suelen depender en gran medida de su estado para determinar cómo tienen que responder a las llamadas a métodos. Esto es particularmente cierto en el caso del motor de la calculadora. Por tanto, una parte importante de la realización del recorrido manual consistirá en asegurarse de que siempre dispongamos de una representación precisa del estado del objeto. Una forma de hacer esto sobre papel es dibujar una tabla con los campos del objeto y sus valores correspondientes (Figura 7.7). Podemos ir escribiendo una nueva línea cada vez, para mantener un registro dinámico de los valores existentes después de cada llamada a método.

Esta técnica hace que resulte muy sencillo volver atrás para comprobar cualquier detalle, en caso de que algo parezca ir mal. También es posible comparar los estados después de dos llamadas al mismo método.

1. Al comenzar el recorrido de `CalcEngine`, documentamos el estado inicial del motor, como se hace en la primera fila de valores de la Figura 7.7. Todos sus campos se inicializan en el constructor. Como ya observamos al hacer el recorrido del método de prueba, la inicialización de los objetos es importante, y podemos tomar nota en este momento de que la inicialización predeterminada es suficiente, particularmente porque el valor predeterminado de `previousOperator` parece no representar ningún operador significativo. Además, esto podría llevarnos a reflexionar sobre si realmente tiene sentido disponer de un operador *previo* antes de encontrarse con el primer operador real en un cálculo. Al anotar estas cuestiones, no tenemos que intentar necesariamente averiguar las respuestas en este mismo momento, sino que esas cuestiones simplemente proporcionan ideas que posteriormente pueden ser aprovechables, cuando descubramos más cosas acerca de la clase.

2. El siguiente paso consiste en comprobar cómo se ve afectado el estado del motor de cálculo al hacerse una llamada a `clear`. Como se muestra en la Figura 7.7, el estado no se ve afectado en este punto, porque `displayValue` ya está configurado con el valor 0. Pero aquí podríamos tomar nota de otra cuestión: ¿por qué este método solo configura el valor de uno

Figura 7.7 Tabulación informal del estado de un objeto.

Llamada a método	displayValue	leftOperand	previousOperator
estado inicial	0	0	' '
clear	0	0	' '
numberPressed(3)	3	0	' '

de los campos? Si se supone que este método implementa una especie de reinicialización, ¿por qué no borrar todos los campos?

3. A continuación, investigamos la llamada a `numberPressed` con un valor de parámetro igual a 3. El método multiplica el valor existente de `displayValue` por 10 y luego le suma el nuevo dígito. Esto modela correctamente el efecto de añadir un nuevo dígito en el extremo derecho de un número existente. El correcto funcionamiento del método depende de que `displayValue` tenga un valor inicial igual a 0 cuando se introduce el primer dígito de un nuevo número, y nuestro análisis del método `clear` nos permite confiar en que esto será así. Por tanto, este método parece correcto.

4. Continuando con el orden de llamadas en el método `testPlus`, a continuación examinamos `plus`. Su primera instrucción llama al método `applyPreviousOperator`. Aquí, tenemos que decidir si seguimos ignorando las llamadas a método anidadas o si interrumpimos el análisis del método actual y vemos qué es lo que el método anidado hace. Echando un rápido vistazo al método `applyPreviousOperator`, podemos ver que es muy corto. Además, está claro que va a modificar el estado del motor, por lo que no vamos a poder seguir documentando los cambios de estado a menos que recorramos manualmente ese método. Por tanto, decidimos seguir la llamada anidada. Es importante recordar de dónde hemos venido, por lo que pondremos una marca en el listado justo dentro del método `plus`, antes de pasar a analizar el método `applyPreviousOperator`. Si el seguir una llamada a método anidada puede conducir con bastante probabilidad a otras llamadas anidadas adicionales, tendremos que usar algo más que una simple marca para poder luego seguir el camino de vuelta hacia el llamante. En este caso, lo mejor es marcar los puntos de llamada con valores numéricos ascendentes, reutilizando los valores anteriores a medida que vamos volviendo de las llamadas.

5. El método `applyPreviousOperator` nos proporciona algunas indicaciones en cuanto a cómo se utiliza el campo `previousOperator`. También parece responder a una de nuestras cuestiones anteriores: si era correcto tener un espacio como valor inicial del operador previo. El método comprueba explícitamente si `previousOperator` contiene un símbolo + o – antes de aplicarlo. Por tanto, si el campo contiene otro valor, no por ello se aplicará una operación incorrecta. Al finalizar este método, el valor de `leftOperand` habrá cambiado, por lo que anotaremos su nuevo valor en la tabla de estado.

6. Volviendo al método `plus`, en él se configuran los valores de los dos campos restantes, por lo que la siguiente fila de la tabla de estados contendrá los siguientes valores:

```
plus 0 3 '+'
```

Podemos continuar el recorrido manual del motor de la calculadora de manera similar, documentando los cambios de estado, intentando comprender su comportamiento y anotando las preguntas que nos vayan surgiendo por el camino. Los siguientes ejercicios le ayudarán a completar el recorrido manual.

Ejercicio 7.25 Complete la tabla de estados basándose en las subsiguientes llamadas que podemos encontrar en el método `testPlus`:

```
numberPressed(4);
equals();
getDisplayValue();
```

Ejercicio 7.26 Al hacer el recorrido del método `equals`, ¿ha podido obtener la misma confianza que obtuvimos en `applyPreviousOperator`, en lo que se refiere al valor predeterminado de `previousOperator`?

Ejercicio 7.27 Haga el recorrido de una llamada a `clear` inmediatamente después de la llamada a `getDisplayValue` al final de su tabla de estados y anote el nuevo estado. ¿Se encuentra el motor en el mismo estado en que estaba después de hacer la llamada anterior a `clear`? Si no es así, ¿qué impacto cree que podría tener esto en los cálculos posteriores?

Ejercicio 7.28 A la luz de la información obtenida durante el recorrido manual, ¿qué cambios cree que habría que realizar en la clase `CalcEngine`? Haga esos cambios en una versión en papel de la clase y luego vuelva a realizar el recorrido manual. No es necesario que haga el recorrido de la clase `CalcEngineTester`; limítese a repetir las acciones de su método `testAll`.

Ejercicio 7.29 Pruebe a realizar un recorrido manual de la siguiente secuencia de llamadas en su versión corregida de la calculadora.

```
clear();
numberPressed(9);
plus();
numberPressed(1);
minus();
numberPressed(4);
equals();
```

¿Cuál debería ser el resultado? ¿Parece comportarse correctamente el motor y dejar la respuesta correcta en `displayValue`?

7.7.3 Recorridos verbales

Otra forma en la que puede utilizarse la técnica de recorridos manuales para localizar errores en un programa es tratar de explicar a otra persona lo que hace una clase o método. Esto funciona de dos maneras completamente distintas:

- La persona a la que le expliquemos el código podría ser capaz de localizar el error por nosotros.
- A menudo podemos encontrar que el simple acto de tratar de explicar con palabras lo que hace un fragmento de código es suficiente para que comprendamos por qué el código está funcionando mal.

Este último efecto es tan común, que a menudo puede ser útil explicarle un fragmento de código a alguien que no tenga ninguna familiaridad con él; no para que *él* sea capaz de encontrar el error, sino para que *nosotros* lo encontremos.

7.8 Instrucciones de impresión

Probablemente la técnica más común utilizada para comprender y depurar programas, incluso por parte de los programadores más experimentados, consiste en anotar temporalmente los métodos mediante una serie de instrucciones de impresión. Las instrucciones de impresión son muy populares porque existen en la mayoría de los lenguajes, porque todo el mundo las tiene a su disposición y porque son fáciles de añadir con cualquier editor. No se necesita ninguna funcionalidad adicional del software o del lenguaje para poder utilizarlas. A medida que se ejecuta un programa, estas instrucciones de impresión adicionales proporcionarán al usuario información acerca de:

- Los métodos que se han invocado.
- Los valores de los parámetros.
- El orden en que se han invocado los métodos.
- Los valores de las variables locales y los campos en ciertos puntos estratégicos.

El Código 7.5 muestra un ejemplo del aspecto que podría tener el método `numberPressed` al añadirle instrucciones de impresión. Dicha información es particularmente útil a la hora de obtener una imagen de cómo cambia el estado de un objeto a medida que se invocan métodos mutadores. Como ayuda adicional, a menudo merece la pena incluir un método de depuración que imprima los valores actuales de los campos de un objeto. El Código 7.6 muestra un método de ese tipo, `reportState`, para la clase `CalcEngine`.

Código 7.5
Un método al que se le han añadido instrucciones de impresión para depuración.

```
/**
 * Se ha pulsado un botón numérico.
 * @param number El número que se ha pulsado.
 */
public void numberPressed(int number)
{
    System.out.println("numberPressed called with: " + number);
    displayValue = displayValue * 10 + number;

    System.out.println("displayValue is: " + displayValue +
                       " at end of numberPressed.");
}
```

Código 7.6
Un método para informar del estado.

```
/**
 * Imprimir los valores de los campos de este objeto.
 * @param where Dónde se produce este estado.
 */
public void reportState(String where)
{
    System.out.println("displayValue: " + displayValue +
                       " leftOperand: " + leftOperand +
                       " previousOperator: " +
                       previousOperator + " at " + where);
}
```

Si cada método de `CalcEngine` contuviera una instrucción de impresión en su punto de entrada y una llamada a `reportState` al final, la Figura 7.8 muestra la salida que podría resultar de una llamada al método `testPlus` de la clase de prueba. (Esta salida se generó mediante una versión del motor de la calculadora que puede encontrar en el proyecto *calculator-engine-print*.) Dicha salida nos permite hacernos una imagen de cómo fluye el control entre los diferentes métodos. Por ejemplo, podemos ver, a partir del orden en que se informa de los valores de estado, que una llamada a `plus` contiene una llamada anidada a `applyPreviousOperator`.

Figura 7.8
Depuración de la salida a partir de una llamada a `testPlus`.

```
clear called
displayValue: 0 leftOperand: 0 previousOperator: at end of clear
numberPressed called with: 3
displayValue: 3 leftOperand: 0 previousOperator: at end of number...
plus called
applyPreviousOperator called
displayValue: 3 leftOperand: 3 previousOperator: at end of apply...
displayValue: 0 leftOperand: 3 previousOperator: + at end of plus
numberPressed called with: 4
displayValue: 4 leftOperand: 3 previousOperator: + at end of
number...
equals called
displayValue: 7 leftOperand: 0 previousOperator: + at end of equals
```

Las instrucciones de impresión pueden ser muy efectivas para ayudarnos a entender programas o localizar errores, pero presentan una serie de desventajas:

- Normalmente, no suele ser muy práctico añadir instrucciones de impresión a todos los métodos de una clase. Por tanto, solo son verdaderamente efectivas si se han introducido las instrucciones de impresión en los métodos correctos.
- Añadir demasiadas instrucciones de impresión puede conducir a una sobrecarga de información. Una gran cantidad de datos de salida puede hacer que resulte difícil identificar aquello que estamos buscando. Las instrucciones de impresión dentro de bucles son una de las principales fuentes de este problema.
- Una vez que han cumplido su propósito, puede resultar tedioso eliminarlas.
- También existe la posibilidad de que habiéndolas eliminado, podamos necesitarlas más adelante. ¡Puede ser muy frustrante tener que introducirlas de nuevo!

Ejercicio 7.30 Abra el proyecto *calculator-engine-print* y complete la adición de instrucciones de impresión a cada método y al constructor.

Ejercicio 7.31 Cree un `CalcEngineTester` en el proyecto y ejecute el método `testAll`. ¿Le ayuda la salida obtenida a identificar dónde reside el problema?

Ejercicio 7.32 ¿Cree que la cantidad de salida producida por la clase `CalcEngine` en la que todos los métodos contienen instrucciones de impresión es demasiado escasa, excesiva o correcta? Si piensa que es

escasa o excesiva, añada o elimine instrucciones de impresión, hasta que le parezca que el nivel de detalle obtenido es el correcto.

Ejercicio 7.33 ¿Cuáles son las ventajas y las desventajas respectivas de utilizar recorridos manuales o instrucciones de impresión para la depuración? Explique su respuesta.

7.8.1 Activación y desactivación de la información de depuración

Si una clase está todavía en desarrollo en el momento de añadir instrucciones de impresión, a menudo no querremos que se muestre la salida cada vez que se utiliza la clase. Lo más conveniente sería encontrar una forma de activar o desactivar la impresión según se requiera. La forma más común de conseguir esto consiste en añadir un campo adicional de depuración de tipo `boolean` a la clase y hacer que la impresión dependa del valor de ese campo. El Código 7.7 ilustra esta idea.

Código 7.7 Forma de controlar si se imprime o no la información de depuración.

```
/**
 * Se ha pulsado un botón numérico.
 * @param number El número que se ha pulsado.
 */
public void numberPressed(int number)
{
    if(debugging) {
        System.out.println("numberPressed called with: " +
                           number);
    }

    displayValue = displayValue * 10 + number;
    if(debugging) {
        reportState();
    }
}
```

Una variante más económica de este mismo tema consiste en sustituir las llamadas directas a instrucciones de impresión por llamadas a un método de impresión especializado que se añade a la clase.[2] El método de impresión solo imprimirá si el campo `debugging` es `true`. De este modo, las llamadas al método de impresión no tendrían que estar condicionadas por una instrucción if. El Código 7.8 ilustra este enfoque. Observe que esta versión presupone que `reportState` comprueba el campo `debugging` por sí mismo o llama también al nuevo método `printDebugging`.

[2] De hecho, podríamos mover este método a una clase especializada de depuración, pero preferimos mantener las cosas simples durante estas explicaciones.

Código 7.8
Un método para imprimir selectivamente información de depuración.

```
/**
 * Se ha pulsado un botón numérico.
 * @param number El número que se ha pulsado.
 */
public void numberPressed(int number)
{
    printDebugging("numberPressed called with: " + number);

    displayValue = displayValue * 10 + number;
    reportState();
}

/**
 * Imprimr la información de depuración solo si
 * debugging es true.
 * @param info La información de depuración.
 */
public void printDebugging(String info)
{
    if(debugging) {
        System.out.println(info);
    }
}
```

Como puede ver en estos experimentos, hace falta algo de práctica para determinar el nivel de detalle más adecuado a la hora de imprimir, si queremos que la salida sea útil. En la práctica, las instrucciones de impresión se añaden a menudo a un solo método cada vez o a unos pocos métodos, cuando tenemos una idea aproximada de en qué área de nuestro programa se oculta el error que estamos buscando.

7.9 Depuradores

En el Capítulo 3, hemos presentado el uso de un depurador con el fin de comprender cómo funciona una aplicación existente y cómo interactúan sus objetos. De forma bastante similar, podemos utilizar el depurador para localizar los errores.

El depurador es, básicamente, una herramienta software que proporciona soporte para realizar un recorrido de un segmento de código. Normalmente, lo que hacemos es definir un punto de interrupción en la instrucción en la que queremos comenzar nuestro recorrido y luego emplear las funciones *Step* o *Step Into* para llevar a cabo ese recorrido paso a paso.

Una ventaja es que el depurador se encarga automáticamente de anotar el estado de cada objeto, por lo que esta técnica es mucho más rápida y menos susceptible a errores que realizar la misma tarea de forma manual. Una desventaja es que los depuradores no suelen mantener un registro permanente de los cambios de estado, de modo que es más difícil ir hacia atrás y comprobar cuál era el estado hace unas cuantas instrucciones.

Los depuradores también suelen proporcionar información acerca de la *secuencia de llamadas* (o *pila de llamadas*) en cada momento. La secuencia de llamadas muestra el nombre del método que contiene la instrucción actual, el nombre del método desde el que se llamó al método actual, el nombre del método desde el que se llamó *a ese otro método*, etc. Por tanto, la secuencia de llamadas contienen un registro de todos los métodos no finalizados que están actualmente activos –de forma similar a lo que hicimos manualmente al realizar nuestro recorrido, poniendo marcas junto a las instrucciones de llamada a métodos.

En BlueJ, la secuencia de llamadas se muestra en la parte izquierda de la ventana del depurador (Figura 7.9). Podemos seleccionar cualquiera de los nombres de métodos existentes en esa secuencia para inspeccionar los valores actuales de las variables locales de dicho método.

Ejercicio 7.34 Utilizando el proyecto *calculator-engine*, establezca un punto de interrupción en la primera línea del método `testPlus` de la clase `CalcEngineTester`. Ejecute este método. Cuando aparezca el depurador, recorra el código paso a paso. Experimente tanto con el botón *Step* como con el botón *Step Into*.

Ejercicio 7.35 *Ejercicio avanzado* En la práctica, probablemente se encuentre con que el intento de Hacker T. Largebrain de programar la clase `CalcEngine` está demasiado lleno de errores como para que merezca la pena corregirlo. En lugar de eso, escriba su propia versión de la clase partiendo de cero. El proyecto *calculator-gui* contiene clases que proporcionan la GUI mostrada en la Figura 7.6. Puede utilizar este pro-yecto como base para su propia implementación de la clase `CalcEngine`. ¡Asegúrese de documentar su clase exhaustivamente y de crear un conjunto completo de pruebas para su implementación, de modo que su experiencia con el código de Hacker no tenga que ser repetida por su sucesor! Asegúrese de emplear clases dedicadas de prueba de unidades para sus pruebas, en lugar de escribir las pruebas en clases estándar; como ha visto, eso permite verificar con mucha mayor facilidad cuáles son los resultados correctos.

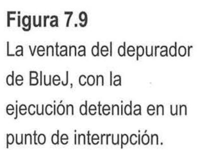
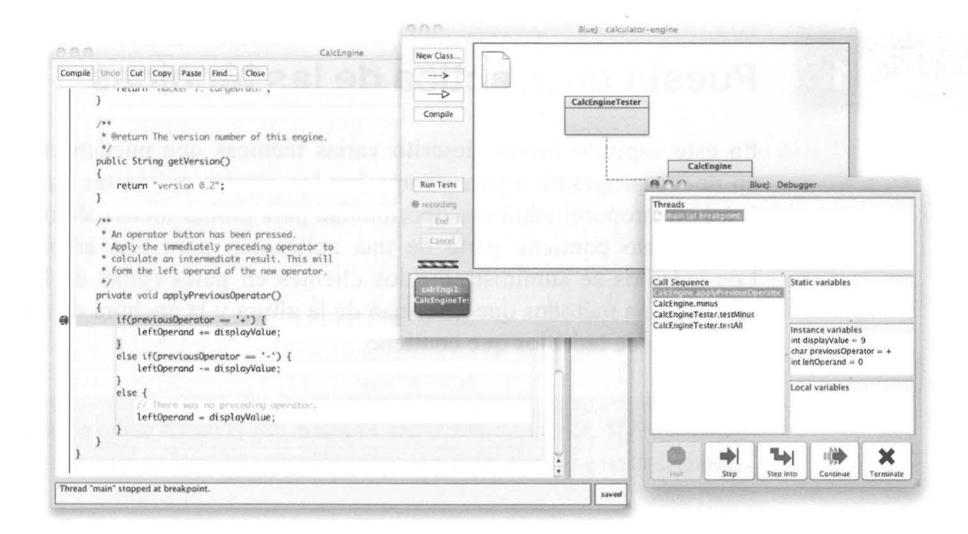


Figura 7.9
La ventana del depurador de BlueJ, con la ejecución detenida en un punto de interrupción.



7.10 Elección de una estrategia de depuración

Hemos visto que existen varias estrategias distintas de realización de pruebas y depuración: recorridos manuales por escrito y verbales, utilización de instrucciones de impresión (bien temporales o permanentes con variables booleanas de activación), pruebas interactivas empleando el banco de objetos, escritura de nuestras propias clases de prueba y utilización de una clase dedicada de prueba de unidades.

En la práctica, utilizaremos distintas estrategias en cada momento. Los recorridos manuales, las instrucciones de impresión y las pruebas interactivas son técnicas útiles para las pruebas iniciales de un código recién escrito, para investigar cómo funciona un segmento de programa o para la depuración. Su ventaja es que son rápidas y fáciles de utilizar, funcionan en cualquier lenguaje de programación y son independientes del entorno (excepto por lo que se refiere a las pruebas interactivas). Su principal desventaja es que las actividades de prueba no son fácilmente repetibles. Esto puede estar bien para la depuración, pero para las pruebas necesitamos algo mejor: hace falta un mecanismo que permita repetirlas fácilmente, con el fin de llevar a cabo pruebas de regresión. La utilización de clases de prueba de unidades tiene la ventaja (una vez que las hemos desarrollado) de que las pruebas pueden reproducirse cualquier número de veces.

Por tanto, la forma de probar de Hacker –escribir su propia clase de prueba– iba en la buena dirección, pero era incorrecta. Ahora sabemos que su problema era que, aunque su clase contenía llamadas a métodos que eran razonables para las pruebas, no incluía ninguna comprobación de los resultados de los métodos, por lo que no podría detectar los fallos en las pruebas. El uso de una clase dedicada de prueba de unidades puede resolver estos problemas.

Ejercicio 7.36 Abra de nuevo su proyecto y añada unos mecanismos de prueba más adecuados, sustituyendo la clase de prueba de Hacker por una clase de prueba de unidades asociada a `CalcEngine`. Añada pruebas similares a las utilizadas por Hacker (y cualquier otra que le parezca útil) e incluya las aserciones correctas.

7.11 Puesta en práctica de las técnicas

En este capítulo hemos descrito varias técnicas que pueden emplearse para comprender un nuevo programa o para comprobar los errores existentes en un programa. El proyecto *bricks* le proporcionará una oportunidad para probar dichas técnicas en un nuevo escenario. El proyecto contiene parte de una aplicación para una empresa que fabrica ladrillos. Los ladrillos se suministran a los clientes en palés (pilas de ladrillos). La clase `Pallet` proporciona métodos que informan de la altura y la anchura de cada palé individual, según el número de ladrillos que contiene.

Ejercicio 7.37 Abra el proyecto *bricks*. Pruébelo. Hay al menos cuatro errores en este proyecto. Vea si los puede localizar y corregir. ¿Qué técnicas ha utilizado para localizar los errores? ¿Qué técnica le ha resultado más útil?

7.12 Resumen

A la hora de escribir software, debemos presuponer que contendrá errores lógicos. Por tanto, es esencial considerar tanto las pruebas como la depuración como actividades normales dentro del proceso de desarrollo. BlueJ es particularmente bueno dando soporte a la prueba interactiva de unidades, tanto de métodos como de clases. También hemos examinado algunas técnicas básicas para automatizar el proceso de pruebas y realizar una depuración simple.

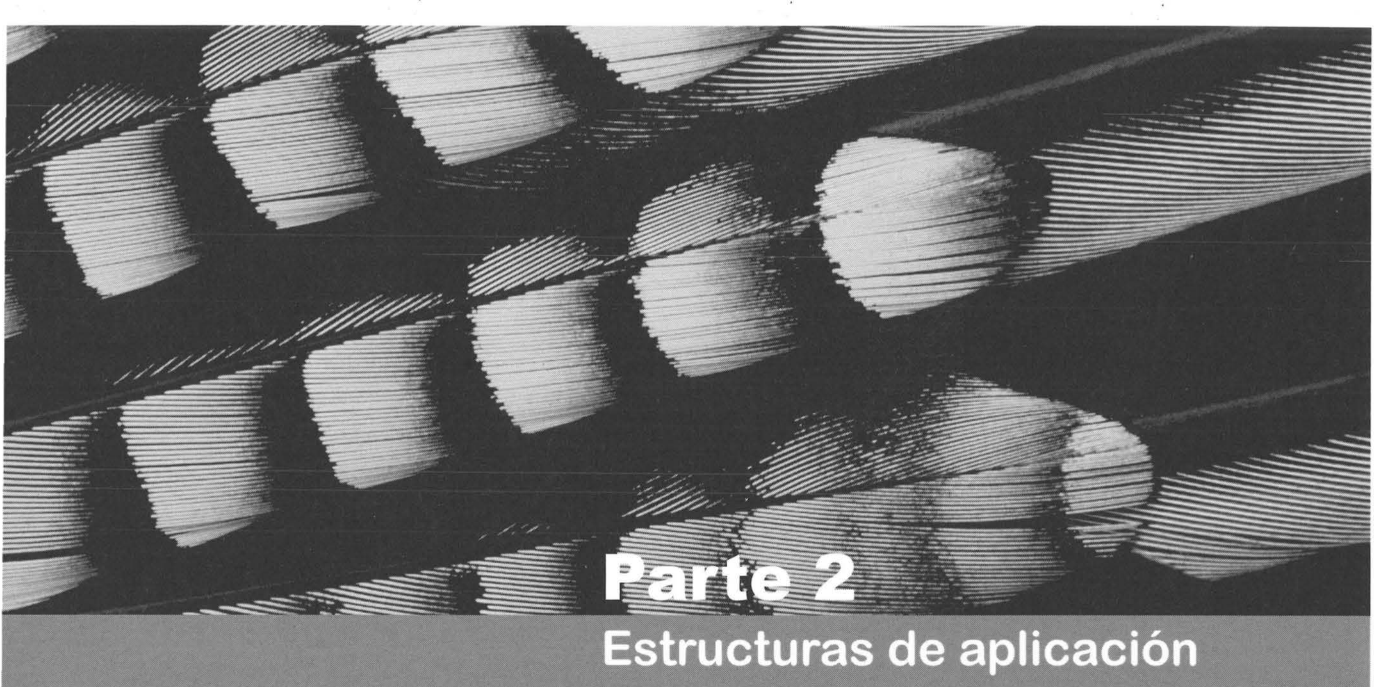
Escribir pruebas JUnit adecuadas para nuestras clases garantizará que los errores se detecten en una fase temprana, y proporcionará una buena indicación de en qué parte del sistema se está produciendo un error, facilitando así en buena medida la correspondiente tarea de depuración.

Términos introducidos en el capítulo

error sintáctico, error lógico, pruebas, depuración, prueba de unidades, JUnit, prueba positiva, prueba negativa, pruebas de regresión, recorrido manual, secuencia de llamadas

Resumen de conceptos

- **pruebas** Las pruebas son la actividad consistente en averiguar si un fragmento de código (un método, una clase o un programa) presenta el comportamiento deseado.
- **depuración** La depuración es el intento de localizar y corregir el origen de un error.
- **prueba positiva** Una prueba positiva es la prueba de un caso que se espera que funcione correctamente.
- **prueba negativa** Una prueba negativa es la prueba de un caso que se espera que falle.
- **aserción** Una aserción es una expresión que indica que establece una condición que esperamos que sea cierta. Si la condición es falsa, decimos que la aserción ha fallado. Esto indica que hay algún error en nuestro programa.
- **banco de pruebas** Un banco de pruebas es un conjunto de objetos en un estado definido que sirven como base para realizar pruebas de unidades..
- **recorrido manual** Un recorrido manual es la actividad consistente en analizar un segmento de código línea a línea mientras que se observan los cambios de estado y otros comportamientos de la aplicación.



Parte 2

Estructuras de aplicación

CAPÍTULO

8 Mejora de la estructura mediante la herencia

Principales conceptos explicados en el capítulo:

- herencia
- sustitución
- subtipos
- variables polimórficas

Estructuras Java explicadas en este capítulo:

`extends`, `super` (en constructor), *cast*, `Object`, *autoboxing*, clases envoltorio

En este capítulo, vamos a presentar algunas estructuras adicionales orientadas a objetos, para mejorar la estructura general de nuestras aplicaciones. Los conceptos principales que emplearemos para diseñar mejores estructuras de programa son la *herencia* y el *polimorfismo*.

Ambos conceptos son fundamentales dentro del marco de la orientación a objetos y veremos posteriormente cómo aparecen de distintas maneras en cada concepto que introduzcamos de aquí en adelante. Sin embargo, no solo son los capítulos siguientes los que dependen fuertemente de estos conceptos: muchas de las estructuras y técnicas explicadas en los capítulos anteriores están influidas por determinados aspectos de la herencia y el polimorfismo, por lo que volveremos a repasar algunas cuestiones presentadas anteriormente y trataremos de obtener una comprensión completa de las interconexiones entre las distintas partes del lenguaje Java.

La herencia es un concepto muy potente que puede utilizarse para dar solución a una gran variedad de problemas. Como siempre, explicaremos los aspectos importantes mediante un ejemplo. En él, presentaremos primero algunos de los problemas que pueden resolverse empleando estructuras de herencia e iremos exponiendo otros usos y ventajas de la herencia y el polimorfismo a medida que vayamos avanzando en el capítulo.

El ejemplo que utilizaremos para presentar estos nuevos conceptos se denomina *network*.

8.1 El ejemplo *network*

El proyecto *network* implementa un prototipo de una pequeña parte de una aplicación de red social como Facebook o Google+. La parte en la que nos concentraremos es la *fuente de noticias* –la lista de mensajes que aparecerá en pantalla cuando un usuario abra la página principal de la red social.

Aquí, comenzaremos de forma simple y poco ambiciosa, con la idea de ampliar y mejorar la aplicación más adelante. Inicialmente, solo tenemos dos tipos de publicaciones en nuestra fuente de noticias; publicaciones de texto (que denominaremos simplemente *mensajes*) y publicaciones fotográficas compuestas por una fotografía y un título.

La parte de la aplicación de la que vamos a hacer aquí un prototipo es el motor que almacena y visualiza estas publicaciones. La funcionalidad que queremos proporcionar con este prototipo debe al menos incluir lo siguiente:

- Debe permitirnos crear publicaciones de texto y fotográficas.
- Las publicaciones de texto están compuestas por un mensaje de longitud arbitraria, que posiblemente ocupa varias líneas. Las publicaciones fotográficas están compuestas por una imagen y un título. Con cada publicación se almacenan algunos detalles adicionales.
- Debe almacenarse esta información de manera permanente, para que pueda utilizarse posteriormente.
- Debe proporcionarse una función de búsqueda que nos permita localizar, por ejemplo, todas las publicaciones de un cierto usuario o todas las fotografías dentro de un cierto de rango de fechas.
- Debe permitir mostrar listas de publicaciones, como la lista de las publicaciones más recientes o una lista de todas las publicaciones de un cierto usuario.
- Debe permitirnos eliminar información.

Los detalles que queremos almacenar para cada publicación de mensaje son:

- El nombre de usuario del autor.
- El texto del mensaje.
- Una marca temporal (instante de la publicación).
- A cuántas personas les ha gustado la publicación.
- Una lista de los comentarios que otros usuarios han hecho acerca de esta publicación.

Los detalles que queremos almacenar para cada publicación fotográfica son:

- El nombre de usuario del autor.
- El nombre del archivo de la imagen que hay que visualizar.
- El título de la fotografía (una línea de texto).
- Una marca temporal (instante de la publicación).
- A cuántas personas les ha gustado la publicación.
- Una lista de los comentarios que otros usuarios han hecho acerca de esta publicación.

8.1.1 El proyecto *network*: clases y objetos

Para implementar la aplicación, primero tenemos que decidir qué clases vamos a utilizar para modelar este problema. En este caso, algunas de las clases son fáciles de identificar. Es bastante sencillo decidir que debe disponer de una clase `MessagePost` para representar las publicaciones de mensajes y una clase `PhotoPost` para representar las publicaciones de fotografías.

Los objetos de estas clases deben entonces encapsular todos los datos que deseemos almacenar acerca de esos objetos (Figura 8.1).

Algunos de estos elementos de datos deberían tener también, probablemente, métodos selectores y mutadores (Figura 8.2).[1] Para nuestros propósitos, no es importante decidir en este momento los detalles exactos de todos los métodos, sino que simplemente queremos tener una primera impresión del diseño de esta aplicación. En esta figura, hemos definido métodos selectores y mutadores para aquellos campos que pueden variar a lo largo del tiempo (como sucede cuando un usuario dice que le gusta o le disgusta una publicación, o cuando se añade un comentario) y suponemos por ahora que los otros campos se configuran

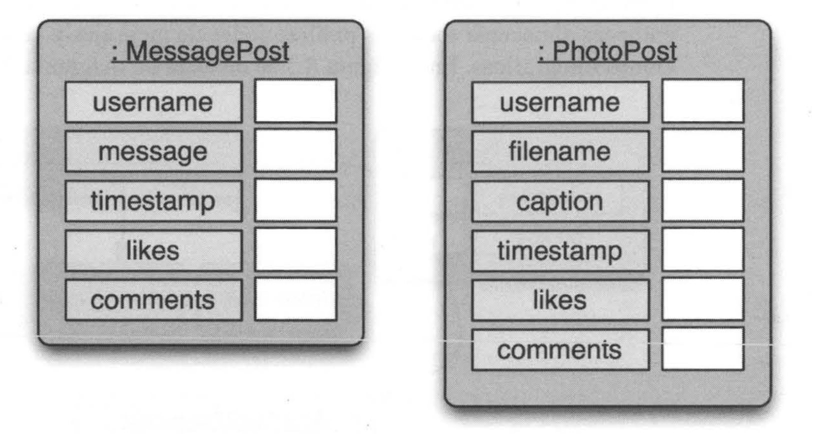


Figura 8.1
Campos de los objetos `MessagePost` y `PhotoPost`.

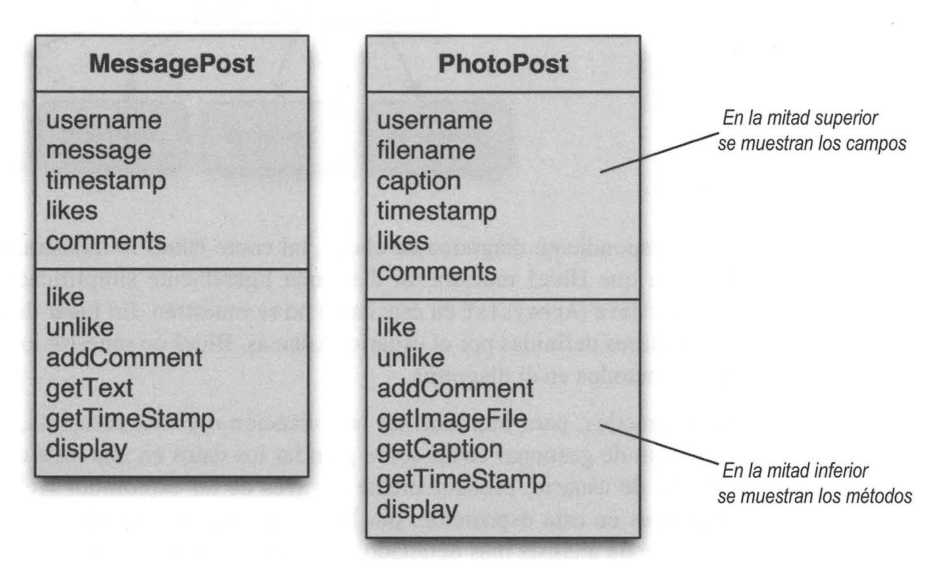


Figura 8.2
Detalles de las clases `MessagePost` y `PhotoPost`.

[1] El estilo de notación para los diagramas de clases utilizados en este libro y en BlueJ es un subconjunto de una notación ampliamente utilizada denominada UML. Aunque no empleamos (ni de lejos) todo lo que UML incluye, hemos intentado utilizar notación UML para las cosas que vamos mostrando. El estilo UML define cómo se muestran los campos y los métodos en un diagrama de clases. La clase se divide en dos partes que muestran (por este orden comenzando por la parte superior) el nombre de la clase, los campos y los métodos.

en el constructor. También hemos añadido un método denominado `display` que mostrará los detalles de un objeto `MessagePost` o `PhotoPost`.

Una vez que hemos definido las clases `MessagePost` y `PhotoPost`, podemos crear tantos objetos publicación como necesitemos –un objeto por cada publicación de mensaje o publicación fotográfica que queramos almacenar. Además de esto necesitamos entonces otro objeto, un objeto que represente la fuente de noticias completa que pueda almacenar una colección de publicaciones de mensajes y fotográficas. Para ello, crearemos una clase denominada `NewsFeed`.

El objeto `NewsFeed` podría él mismo contener dos objetos colección (por ejemplo, de tipos `ArrayList <MessagePost>` y `ArrayList<PhotoPost>`). Una de estas colecciones puede entonces almacenar todas las publicaciones de mensajes y la otra almacenar todas las publicaciones fotográficas. En la Figura 8.3 se muestra un diagrama de objetos para este modelo.

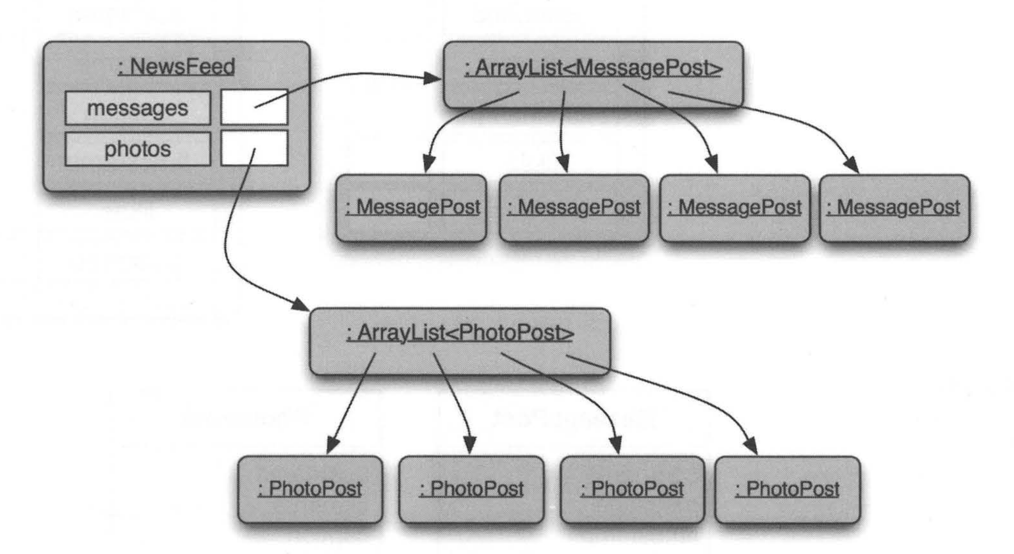


Figura 8.3
Objetos de la aplicación *network*.

El correspondiente diagrama de clases, tal como BlueJ lo muestra, se ilustra en la Figura 8.4. Observe que BlueJ muestra un diagrama ligeramente simplificado: las clases de la librería estándar Java (`ArrayList` en este caso) no se muestran. En lugar de ello, el diagrama se centra en las clases definidas por el usuario. Además, BlueJ no muestra los nombres de los campos ni de los métodos en el diagrama.

En la práctica, para implementar la aplicación *network* completa, necesitaríamos más clases con el fin de gestionar cosas como guardar los datos en una base de datos y proporcionar una interfaz de usuario, probablemente a través de un explorador web. Estas clases no son muy relevantes en esta exposición, por lo que evitaremos describirlas por ahora y nos concentraremos en un análisis más detallado de las clases fundamentales que aquí hemos mencionado.

8.1.2 Código fuente de *network*

Hasta ahora, el diseño de las tres clases actuales (`MessagePost`, `PhotoPost` y `NewsFeed`) ha sido muy sencillo. Traducir estas ideas a código Java también es igual de sencillo. El Código 8.1 muestra el código fuente de la clase `MessagePost`. Define los campos apropiados, configura en

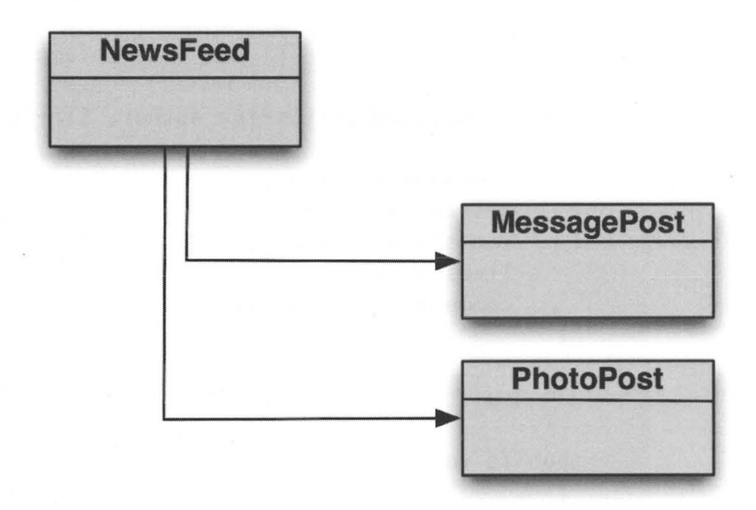


Figura 8.4
Diagrama de clases BlueJ para *network*.

su constructor todos los datos que se espera que no cambien a lo largo del tiempo y proporciona métodos selectores y mutadores en los casos apropiados. También implementa el método `display`, para mostrar la publicación en el terminal de texto.

Código 8.1
Código fuente de la clase `MessagePost`.

```
import java.util.ArrayList;

/**
 * Esta clase almacena información acerca de una publicación
 * en una red social. La parte principal de la publicación está
 * compuesta por un mensaje de texto (posiblemente de varias
 * líneas). También se almacenan otros datos como el autor y la
 * fecha.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 0.1
 */
public class MessagePost
{
    private String username; // nombre de usuario del autor
                             // de la publicación
    private String message;  // un mensaje de varias líneas,
                             // de longitud arbitraria.
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Constructor para los objetos de la clase MessagePost.
     *
     * @param author    El nombre de usuario del autor de
     *                  esta publicación.
```

Código 8.1
(*continuación*)
Código fuente de la clase `MessagePost`.

```
     * @param text      El texto de esta publicación.
     */
    public MessagePost(String author, String text)
    {
        username = author;
        message = text;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }
    /**
     * Registrar un indicación más "Me gusta" de un usuario.
     */
    public void like()
    {
        likes++;
    }

    /**
     * Registrar que un usuario ha decidido retirar su voto
     * "Me gusta" a esta publicación.
     */
    public void unlike()
    {
        if (likes > 0) {
            likes--;
        }
    }

    /**
     * Añadir un comentario a esta publicación.
     *
     * @param text  El nuevo comentario que hay que añadir.
     */
    public void addComment(String text)
    {
        comments.add(text);
    }

    /**
     * Devolver el texto de esta publicación.
     *
     * @return El texto de esta publicación.
     */
    public String getText()
    {
        return message;
    }
```

Código 8.1
(*continuación*)
Código fuente de la clase `MessagePost`.

```
/**
 * Devolver el instante de creación de esta publicación.
 *
 * @return El instante de creación de la publicación, como valor
 *         de la hora del sistema.
 */
public long getTimeStamp()
{
    return timestamp;
}

/**
 * Mostrar los detalles de esta publicación.
 *
 * (Actualmente: imprimir en el terminal de texto.
 * Esto sirve para simular por ahora la visualización
 * en un explorador web.)
 */
public void display()
{
    System.out.println(username);
    System.out.println(message);
    System.out.print(timeString(timestamp));
    if(likes > 0) {
        System.out.println("  -  " + likes + " people like this.");
    }
    else {
        System.out.println();
    }
    if(comments.isEmpty()) {
        System.out.println("   No comments.");
    }
    else {
        System.out.println("   " + comments.size() +
                           " comment(s). Click here to view.");
    }
}

/**
 * Crear una cadena que describa un instante temporal en el
 * pasado en términos relativos, como por ejemplo
 * "30 seconds ago" (hace 30 segundos) o "7 minutes ago"
 * (hace 7 minutos).
 * Actualmente, solo se emplean los segundos y los minutos
 * para esta cadena
 *
 * @param time  El valor temporal que hay que convertir
 *              (en milisegundos del sistema)
```

Código 8.1
(*continuación*)
Código fuente de la clase `MessagePost`.

```
     * @return      Una cadena temporal relativa para el instante
     *             temporal especificado.
     */
    private String timeString(long time)
    {
        long current = System.currentTimeMillis();
        long pastMillis = current - time; // tiempo transcurrido
                                           // en milisegundos
        long seconds = pastMillis / 1000;
        long minutes = seconds / 60;
        if(minutes > 0) {
            return minutes + " minutes ago";
        }
        else {
            return seconds + " seconds ago";
        }
    }
}
```

Merece la pena analizar brevemente algunos detalles:

- Se han hecho algunas simplificaciones. Por ejemplo, los comentarios de la publicación se almacenan como cadenas de caracteres. En una versión más completa, probablemente utilizaríamos una clase personalizada para los comentarios, ya que estos tienen también detalles adicionales, como un autor y un instante temporal. La cuenta de votos "Me gusta" se almacena como un entero simple. Actualmente, no guardamos la información de qué usuarios han votado por una publicación. Aunque estas simplificaciones hacen que nuestro prototipo esté incompleto, no son relevantes para las explicaciones fundamentales que estamos exponiendo, por lo que dejaremos las cosas así por el momento.
- La marca temporal se almacena como un único número de tipo `long`. Esto es lo que se suele hacer en la práctica. Podemos obtener fácilmente la hora del sistema en Java como un valor de tipo `long`, en milisegundos. También hemos escrito un método corto, denominado `timeString`, para convertir este número en una cadena de caracteres temporal relativa, como por ejemplo "*5 minutes ago*" ("hace 5 minutos"). En nuestra aplicación final, el sistema tendría que utilizar tiempo real en lugar de tiempo del sistema, pero de nuevo el tiempo del sistema es suficientemente bueno para nuestro prototipo, al menos por el momento.

Observe que no pretendemos todavía hacer que la implementación sea completa en ningún sentido. Sirve simplemente para hacernos una idea del aspecto que podría tener una clase como esta. Emplearemos este prototipo como base para nuestras posteriores explicaciones acerca de la herencia.

Ahora vamos a comparar el código fuente de `MessagePost` con el de la clase `PhotoPost`, mostrado en el Código 8.2. Examinando ambas clases, observamos rápidamente que son muy similares. Esto no es sorprendente, ya que su propósito es parecido: ambas se usan para almacenar información acerca de las publicaciones existentes en la fuente de noticias y los diferentes tipos de publicación tienen muchas cosas en común. Solo difieren en los detalles, como por ejemplo en algunos de sus campos, en los métodos selectores correspondientes y en los cuerpos del método `display`.

Código 8.2
Código fuente de la clase `PhotoPost`.

```
import java.util.ArrayList;

/**
 * Esta clase almacena información acerca de una publicación
 * en una red social. La parte principal de la publicación está
 * compuesta por una fotografía y un título. También se almacenan
 * otros datos como el autor y el instante de publicación.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 0.1
 */
public class PhotoPost
{
    private String username;  // nombre de usuario del autor de
                              // la publicación
    private String filename;  // nombre del archivo de imagen
    private String caption;   // título de la imagen de una
                              // sola línea
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Constructor para objetos de la clase PhotoPost.
     *
     * @param author    El nombre de usuario del autor
     *                  de esta publicación.
     * @param filename  El nombre de archivo de la imagen
     *                  de esta publicación.
     * @param caption   Un título para la imagen.
     */
    public PhotoPost(String author, String filename, String caption)
    {
        username = author;
        this.filename = filename;
        this.caption = caption;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }

    /**
     * Registrar una indicación "Me gusta" de un usuario.
     */
    public void like()
    {
        likes++;
    }
```

Código 8.2
(*continuación*)
Código fuente de la clase `PhotoPost`.

```
    /**
     * * Registrar que un usuario ha decidido retirar su voto
     * "Me gusta" a esta publicación.
     */
    public void unlike()
    {
        if (likes > 0) {
            likes--;
        }
    }

    /**
     * Añadir un comentario a esta publicación.
     *
     * @param text  El nuevo comentario que hay que añadir.
     */
    public void addComment(String text)
    {
        comments.add(text);
    }

    /**
     * Devolver el nombre de archivo de la imagen
     * de esta publicación.
     *
     * @return El nombre de archivo de imagen de esta publicación.
     */
    public String getImageFile()
    {
        return filename;
    }

    /**
     * Devolver el título de la imagen de esta publicación.
     *
     * @return El título de la imagen.
     */
    public String getCaption()
    {
        return caption;
    }

    /**
     * Devolver el instante de creación de esta publicación.
     *
     * @return El instante de creación de la publicación, como valor
     *         de la hora del sistema.
     */
```

Código 8.2
(*continuación*)
Código fuente de la clase `PhotoPost`.

```
    public long getTimeStamp()
    {
        return timestamp;
    }

    /**
     * Mostrar los detalles de esta publicación.
     *
     * (Actualmente: imprimir en el terminal de texto.
     * Esto sirve para simular por ahora la visualización
     * en un explorador web.)
     */
    public void display()
    {
        System.out.println(username);
        System.out.println("  [" + filename + "]");
        System.out.println("  " + caption);
        System.out.print(timeString(timestamp));
        if(likes > 0) {
            System.out.println(" — " + likes + " people like this.");
        }
        else {
            System.out.println();
        }
        if(comments.isEmpty()) {
            System.out.println("   No comments.");
        }
        else {
            System.out.println("   " + comments.size() +
                               " comment(s). Click here to view.");
        }
    }

    /**
     * Crear una cadena que describa un instante temporal en el
     * pasado en términos relativos, como por ejemplo
     * "30 seconds ago" (hace 30 segundos) o "7 minutes ago"
     * (hace 7 minutos).
     *
     * Actualmente, solo se emplean los segundos y los minutos
     * para esta cadena
     *
     * @param time  El valor temporal que hay que convertir
     *              (en milisegundos del sistema)
     * @return      Una cadena temporal relativa para el instante
     *              temporal especificado.
     *
     */
```

Código 8.2
(*continuación*)
Código fuente de la clase `PhotoPost`.

```
    private String timeString(long time)
    {
        long current = System.currentTimeMillis();
        long pastMillis = current - time;  // tiempo transcurrido
                                           // en  milisegundos
        long seconds = pastMillis / 1000;
        long minutes = seconds / 60;
        if(minutes > 0) {
            return minutes + " minutes ago";
        }
        else {
            return seconds + " seconds ago";
        }
    }
}
```

A continuación, examinemos el código fuente de la clase `NewsFeed` (Código 8.3). También es muy simple. Define dos listas (cada una de ellas basada en la clase `ArrayList`) para almacenar la colección de publicaciones de mensajes y la colección de publicaciones fotográficas. Es en el constructor donde se crean las listas vacías. Después, proporciona dos métodos para añadir elementos: uno para añadir publicaciones de mensajes y otro para añadir publicaciones fotográficas. El último método, denominado `show`, imprime en el terminal de texto una lista de todos las publicaciones de mensajes y fotográficas.

Código 8.3
Código fuente de la clase `NewsFeed`.

```
import java.util.ArrayList;
/**
 * La clase NewsFeed almacena publicaciones para la fuente de
 * noticias en una aplicación de red social (como FaceBook o
 * Google+).
 *
 * La visualización de las publicaciones se simula actualmente
 * imprimiendo los detalles en el terminal. (Más adelante, debería
 * mostrarse en un explorador web.)
 *
 * Esta versión no guarda los datos en disco y no proporciona
 * ninguna función de búsqueda ni de ordenación.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 0.1
 */
public class NewsFeed
{
    private ArrayList<MessagePost> messages;
    private ArrayList<PhotoPost> photos;
```

Código 8.3
(*continuación*)
Código fuente de la clase `NewsFeed`.

```
    /**
     * Construye una fuente de noticias vacía.
     */
    public NewsFeed()
    {
        messages = new ArrayList<MessagePost>();
        photos = new ArrayList<PhotoPost>();
    }

    /**
     * Añadir una publicación de texto a la fuente de noticias.
     * @param text  La publicación de texto que hay que añadir.
     */
    public void addMessagePost(MessagePost message)
    {
        messages.add(message);
    }

    /**
     * Añadir una publicación fotográfica a la fuente de noticias.
     *
     * @param photo  La publicación fotográfica que hay que añadir.
     */
    public void addPhotoPost(PhotoPost photo)
    {
        photos.add(photo);
    }

    /**
     * Mostrar la fuente de noticias. Actualmente: imprimir los
     * detalles en el terminal. (Para más adelante: sustituir esto
     * por una visualización en un explorador web.)
     */
    public void show()
    {
        // Mostrar todas las publicaciones de texto
        for(MessagePost message : messages) {
            message.display();
            System.out.println(); // línea vacía entre publicaciones
        }

        // Mostrar todas las fotografías
        for(PhotoPost photo : photos) {
            photo.display();
            System.out.println(); // línea vacía entre fotografías
        }
    }
}
```

Esta no es, en modo alguno, una aplicación completa. No tiene interfaz de usuario (por lo que no se podría utilizar fuera de BlueJ) y los datos introducidos no se almacenan en el sistema de archivos ni en una base de datos. Esto significa que todos los datos introducidos se perderán cada vez que la aplicación termine. No hay tampoco funciones para ordenar la lista de publicaciones mostrada, por ejemplo, por fecha y hora o por relevancia. Actualmente, siempre obtenemos primero los mensajes en el orden en el que fueron introducidos, seguidos de las fotografías. Asimismo, las funciones para introducir y editar los datos, así como para buscar datos y visualizarlos, no son suficientemente flexibles, si las comparamos con lo que se esperaría de un programa real.

Sin embargo, esto no importa en nuestro contexto. Posteriormente, podemos trabajar en intentar mejorar la aplicación. Lo importante es que la estructura básica está ya disponible y funciona. Esto nos basta para explicar los problemas de diseño y las posibles mejoras.

Ejercicio 8.1 Abra el proyecto *network-v1*. Contiene las clases exactamente como las hemos presentado aquí. Cree algunos objetos `MessagePost` y `PhotoPost`. Cree un objeto `NewsFeed`. Introduzca las publicaciones en la fuente de noticias y luego visualice el contenido de la fuente.

Ejercicio 8.2 Intente hacer lo siguiente: cree un objeto `MessagePost`, introdúzcalo en la fuente de noticias y visualice la fuente de noticias. Verá que la publicación no tiene ningún comentario asociado. Añada un comentario al objeto `MessagePost` que hay en el banco de objetos (el que ha introducido en la fuente de noticias). Cuando ahora visualice de nuevo la fuente de noticias, ¿tendrá un comentario asociado la publicación que aparece allí? Compruébelo. Explique el comportamiento que observe.

8.1.3 Análisis de la aplicación *network*

Aunque nuestra aplicación no está completa todavía, ya hemos hecho la parte más importante. Hemos definido el núcleo de la aplicación –la estructura de datos que almacena la información esencial.

Todo ha sido bastante sencillo hasta el momento, por lo que ahora podríamos continuar y diseñar lo que nos falta. Sin embargo, antes de hacer eso, vamos a analizar la calidad de la solución que tenemos en este momento.

Hay varios problemas fundamentales con nuestra actual solución. El más obvio es la *duplicación de código*.

Ya hemos dicho que las clases `MessagePost` y `PhotoPost` son muy similares. De hecho, la mayor parte del código fuente de esas clases es idéntico, con solo unas pocas diferencias. Ya hemos mencionado los problemas asociados con la duplicación de código en el Capítulo 6. Aparte del molesto hecho de que tengamos que escribir todo dos veces (o bien copiar y pegar y luego recorrer el código para implementar todas las diferencias), a menudo se presentan problemas con el mantenimiento del código duplicado. Muchos posibles cambios tendrían que efectuarse dos veces. Por ejemplo, si se cambiara el tipo de la lista de comentarios de `ArrayList<String>` a `ArrayList<Comment>` (para poder almacenar más detalles), este cambio tendría que hacerse una vez en la clase `MessagePost` y otra en la clase `PhotoPost`. Además, asociado con el problema del mantenimiento de la duplicación del código está siempre el peligro de introducir errores, porque el programador de mantenimiento podría no darse cuenta de que hace falta un cambio idéntico en una segunda (o en una tercera) ubicación.

Hay otro lugar en el que nos encontramos con código duplicado: la clase `NewsFeed`. Podemos ver que todo en esa clase se hace dos veces, una vez para las publicaciones de mensajes y otra para las publicaciones fotográficas. La clase define dos variables de lista, luego crea dos objetos lista, define dos métodos `add` y tiene dos bloques de código casi idénticos en el método `show` para imprimir las listas.

Los problemas con esta duplicación se hacen patentes cuando analizamos lo que habría que hacer para añadir otro tipo de publicación a este programa. Imagine que quisiéramos almacenar no solo mensajes de texto y fotografías, sino también publicaciones relativas a la actividad. Las publicaciones relativas a la actividad pueden ser generadas automáticamente y nos informan de la actividad de uno de nuestros contactos, como por ejemplo "*Fred has changed his profile picture*" ("Fred ha cambiado su imagen de perfil") o "*Ava is now friends with Feena.*" ("Ava es ahora amiga de Feena"). Las publicaciones de actividad parecen lo suficientemente similares a las otras como para que resulte fácil modificar la aplicación para incluirlas. Lo que haríamos sería introducir otra clase, `ActivityPost`, y escribir esencialmente una tercera versión del código fuente que ya tenemos en las clases `MessagePost` y `PhotoPost`. Después, tendríamos que recorrer la clase `NewsFeed` y añadir otra variable de lista, otro objeto de lista, otro método `add` y otro bucle en el método `show`.

Si introdujéramos un cuarto tipo de publicación, entonces tendríamos que volver a hacer lo mismo. Cuanto más repitamos el proceso, mayor se hará el problema de la duplicación de código y más difícil será hacer cambios posteriormente. El que nos sintamos incómodos con una situación como esta suele ser un buen indicador de que debe existir una solución alternativa. Para este caso concreto, podemos encontrar dicha solución en los lenguajes orientados a objetos, que proporcionan una característica distintiva que tiene un gran impacto en los programas que manejan conjuntos de clases similares. En las siguientes secciones vamos a presentar esta característica, que se denomina *herencia*.

8.2 Utilización de la herencia

Concepto:

La **herencia** nos permite definir una clase como ampliación de otra.

La herencia es un mecanismo que nos proporciona una solución a nuestro problema de la duplicación. La idea es simple: en lugar de definir las clases `MessagePost` y `PhotoPost` de forma completamente independiente, definimos primero una clase que contiene todo lo que esas dos clases tienen en común. A esa clase la denominaremos `Post` y representará publicaciones genéricas. A continuación, podemos declarar que un `MessagePost` es un `Post` y que un `PhotoPost` es un `Post`. Por último, añadiremos a la clase `MessagePost` los detalles adicionales necesarios para una publicación de mensaje y añadiremos a la clase `PhotoPost` los detalles correspondientes de una publicación fotográfica. La ventaja esencial de esta técnica es que solo es necesario describir las características comunes una vez.

La Figura 8.5 muestra un diagrama de clases para esta nueva estructura. En la parte superior, se muestra la clase `Post`, que define todos los campos y métodos que son comunes a todas las publicaciones (mensajes y fotografías). Debajo de la clase `Post` aparecen las clases `MessagePost` y `PhotoPost`, que solo almacenan aquellos campos y métodos que son distintivos de cada clase particular.

Esta nueva característica de la programación orientada a objetos requiere alguna nueva terminología. En una situación como esta, decimos que la clase `MessagePost` *hereda de* la clase `Post`. La clase `PhotoPost` también hereda de `Post`. En la jerga de los programas Java,

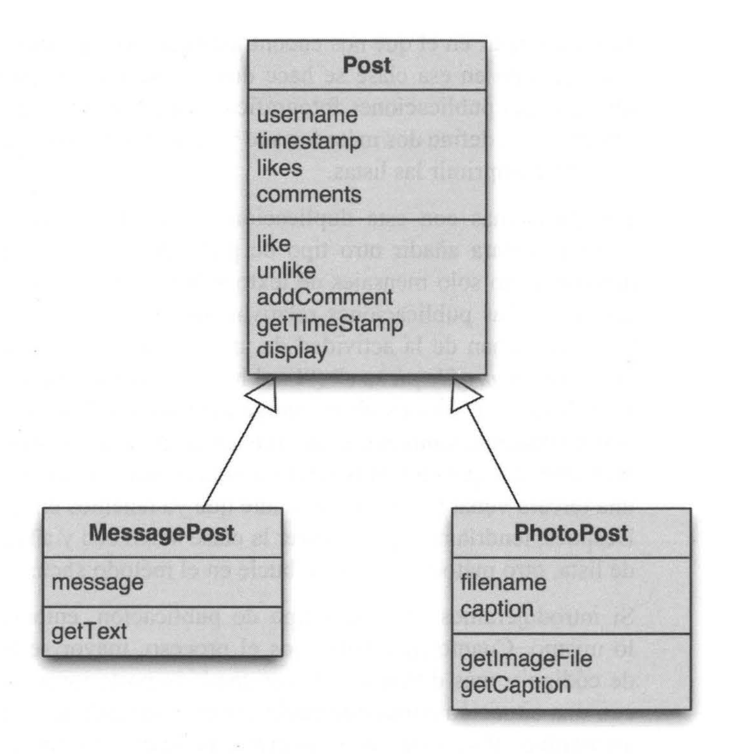


Figura 8.5 `MessagePost` y `PhotoPost` heredando de `Post`.

Concepto:

Llamamos **superclase** a toda clase que es ampliada por otra clase.

Concepto:

Llamamos **subclase** a una clase que amplia (hereda de) otra clase. La subclase hereda todos los campos y métodos de su superclase.

podría utilizarse la expresión "la clase `MessagePost` *amplía* la clase `Post`", porque Java utiliza una palabra clave `extends` (ampliar) para definir la relación de herencia (como veremos enseguida). Las flechas en el diagrama de clases (que usualmente se dibujan con cabezas de flecha huecas) representan la relación de herencia.

La clase `Post` (la clase de la que las otras heredan) es la *clase padre* o *superclase*. Las clases herederas (en este ejemplo `MessagePost` y `PhotoPost`) se denominan *clases hija* o *subclases*. En este libro, utilizaremos los términos "superclase" y "subclase" para hacer referencia a las clases que mantienen una relación de herencia.

La herencia se denomina en ocasiones relación *es-un*. La razón es que una subclase es una especialización de una superclase. Podemos decir que "una publicación de mensaje *es una* publicación" y que "una publicación fotográfica *es una* publicación".

El propósito de utilizar la herencia es ahora bastante obvio. La instancias de `MessagePost` tendrán ahora todos los campos definidos en la clase `MessagePost` *y* en la clase `Post` (`MessagePost` hereda los campos de `Post`). Las instancias de `PhotoPost` tendrán todos los campos definidos en `PhotoPost` y en `Post`. Por tanto, conseguimos lo mismo que con nuestra anterior solución, pero solo necesitamos definir los campos `username`, `timestamp`, `likes` y `comments` una única vez, sin por ello dejar de poder utilizarlos en dos lugares distintos.

Lo mismo cabe decir de los métodos: las instancias de las subclases disponen de todos los métodos definidos tanto en la superclase como en la subclase. En general, podemos decir que: como una publicación de mensaje es una publicación, un objeto publicación-de-mensaje dispondrá de todo aquello de lo que disponga una publicación y de más cosas. Y como una publicación fotográfica es también una publicación, dispondrá de todo aquello de lo que disponga una publicación y de otras cosas adicionales.

Por tanto, la herencia nos permite crear dos clases que son bastante similares al mismo tiempo que evitamos tener que escribir las partes idénticas dos veces. La herencia tiene algunas otras ventajas, de las que hablaremos más adelante. No obstante, vamos a echar otro vistazo más general a las jerarquías de herencia.

8.3 Jerarquías de herencia

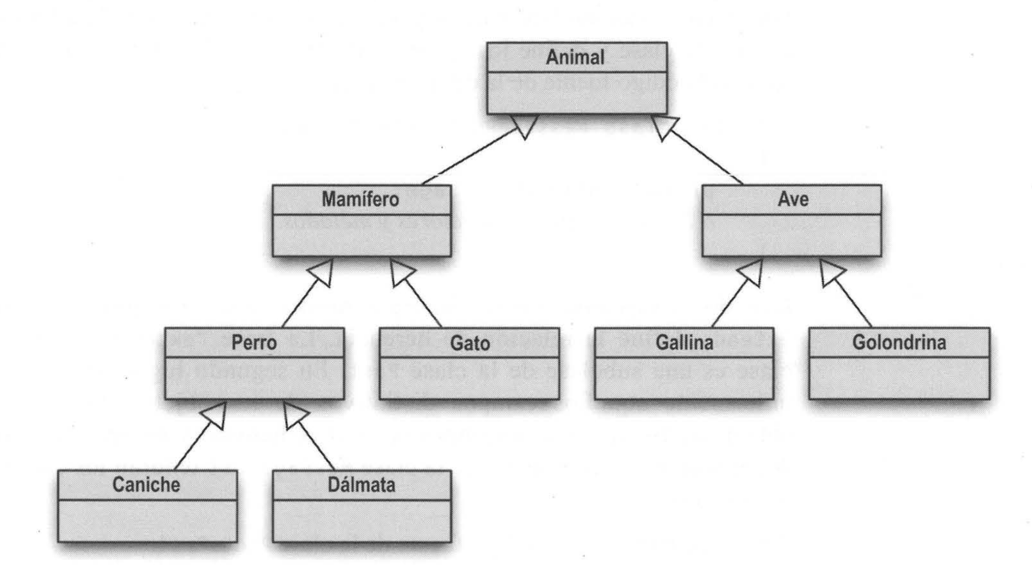
Concepto:

Las clases que están vinculadas por relaciones de herencia forman una **jerarquía de herencia**.

La herencia puede utilizarse de forma mucho más general de lo que se ha mostrado en el ejemplo anterior. Puede haber más de dos subclases heredando de la misma superclase, y una subclase puede, a su vez, actuar como superclase de otras subclases. Las clases formarán entonces lo que se denomina una *jerarquía de herencia.*

El ejemplo más conocido de jerarquía de herencia sea probablemente la clasificación de las especies realizada por los biólogos. En la Figura 8.6 se muestra una pequeña parte. Podemos ver que un caniche es un perro, que a su vez es un mamífero, que a su vez es un animal.

Figura 8.6
Un ejemplo de una jerarquía de herencia.



Sabemos algunas cosas acerca de los caniches –por ejemplo, que están vivos, que pueden ladrar, que comen carne y que son vivíparos. Si examinamos el asunto con más detalle, vemos que algunas de esas cosas no las sabemos porque sean caniches, sino porque son perros, porque son mamíferos o porque son animales. Una instancia de la clase `Caniche` (un caniche real) tendrá todas las características de un caniche, de un perro, de un mamífero y de un animal, porque un caniche es un perro, que a su vez es un mamífero, etc.

El principio es muy simple: la herencia es una técnica de abstracción que nos permite categorizar las clases de objetos usando ciertos criterios y nos ayuda a especificar las características de esas clases.

Ejercicio 8.3 Dibuje una jerarquía de herencia para las personas de su lugar de estudio o de trabajo. Por ejemplo, si es usted un estudiante universitario, entonces su universidad probablemente tenga estudiantes (de primer curso, de segundo curso, ...), profesores, tutores, personal administrativo, etc.

8.4 Herencia en Java

Antes de exponer más detalles acerca de la herencia, vamos a examinar cómo se expresa la herencia en el lenguaje Java. He aquí un segmento del código fuente de la clase `Post`:

```
public class Post
{
    private String username;  // nombre de usuario del autor
                              // de la publicación
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;
    // Se omiten los constructores y métodos.
}
```

Por el momento, no hay nada especial acerca de esta clase. Comienza con una definición normal de clase y define los campos de `Post` de la forma habitual. A continuación, examinamos el código fuente de la clase `MessagePost`:

```
public class MessagePost extends Post
{
    private String message;
    // Se omiten los constructores y métodos.
}
```

Hay dos cosas que merece la pena resaltar aquí. En primer lugar, la palabra clave `extends` define la relación de herencia. La frase "`extends Post`" especifica que esta clase es una subclase de la clase `Post`. En segundo lugar, la clase `MessagePost` define únicamente aquellos campos distintivos de los objetos `MessagePost` (en este caso, solamente `message`). Los campos de `Post` se heredan y no necesitan ser enumerados aquí. A pesar de ello, los objetos de la clase `MessagePost` tendrán los campos correspondientes a `username`, `timestamp`, etc.

Ahora, examinemos el código fuente de la clase `PhotoPost`:

```
public class PhotoPost extends Post
{
    private String filename;
    private String caption;
    // Se omiten los constructores y métodos.
}
```

Esta clase sigue el mismo patrón que la clase `MessagePost`. Utiliza la palabra clave `extends` para definirse a sí misma como subclase de `Post` y luego define sus propios campos adicionales.

8.4.1 Herencia y derechos de acceso

Para los objetos de otras clases, los objetos `MessagePost` o `PhotoPost` tienen la misma apariencia que cualquier otro tipo de objeto. En consecuencia, los miembros definidos

como `public` en la superclase o en la subclase serán accesibles para los objetos de otras clases, pero los miembros definidos como `private` serán inaccesibles.

De hecho, la regla acerca de la privacidad también se aplica entre una subclase y su superclase: una subclase no puede acceder a los miembros privados de su superclase. De aquí se deduce que si un método de la subclase necesita acceder o modificar campos privados de su superclase, entonces la superclase tendrá que proporcionar los métodos selectores y/o mutadores apropiados. Sin embargo, un objeto de una subclase sí que puede invocar cualquier método público definido en su superclase, como si estuviera definido localmente en la subclase –no hace falta ninguna variable, porque todos los métodos forman parte del mismo objeto.

Volveremos sobre este tema de los derechos de acceso entre las superclases y las subclases en el Capítulo 9, cuando presentemos el modificador `protected`.

Ejercicio 8.4 Abra el proyecto *network-v2.* Este proyecto contiene una versión de la aplicación *network*, reescrita para usar la herencia. Como hemos descrito anteriormente. Observe que el diagrama de clases muestra la relación de herencia. Abra el código fuente de la clase `MessagePost` y elimine la frase "`extends Post`". Cierre el editor. ¿Qué cambios observa en el diagrama de clases? Añada de nuevo la frase "`extends Post`".

Ejercicio 8.5 Cree un objeto `MessagePost`. Invoque algunos de sus métodos. ¿Se pueden invocar los métodos heredados (por ejemplo, `addComment`)? ¿Qué es lo que puede observar acerca de los métodos heredados?

Ejercicio 8.6 Para ilustrar que una subclase puede acceder a los elementos no privados de su superclase sin necesidad de utilizar ninguna sintaxis especial, trate de ejecutar la siguiente modificación ligeramente artificial de las clases `MessagePost` y `Post`. Cree un método denominado `printShortSummary` en la clase `MessagePost`. Su tarea consiste en imprimir simplemente la frase "Message post from *NAME*" ("Publicación de mensaje de *NAME*), donde *NAME* debería mostrar el nombre del autor. Sin embargo, puesto que el campo `username` es privado en la clase `Post`, será necesario añadir un método público `getUserName` a `Post`. Invoque este método desde `printShortSummary` para acceder al nombre, con el fin de imprimirlo. Recuerde que no se necesita ninguna sintaxis especial cuando una subclase invoca un método de su superclase. Pruebe su solución creando un objeto `MessagePost`. Implemente un método similar en la clase `PhotoPost`.

8.4.2 Herencia e inicialización

Cuando creamos un objeto, el constructor de dicho objeto se encarga de inicializar todos los campos del objeto en algún estado razonable. Vamos a tratar de examinar con más detalle cómo se hace esto en las clases que heredan de otras clases.

Cuando creamos un objeto `MessagePost`, pasamos dos parámetros al constructor de la publicación de mensaje: el nombre del autor y el texto del mensaje. Uno de estos contiene un valor para un campo definido en la clase `Post`, mientras que el otro contiene un valor para un campo definido en la clase `MessagePost`. Todos estos campos deben ser correctamente inicializados, y el Código 8.4 muestra los segmentos de código que se utilizan en Java para conseguir esto.

Código 8.4
Inicialización de los campos de la subclase y la superclase.

```
public class Post
{
    private String username;  // nombre de usuario del autor
                              // de la publicación.
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Constructor para los objetos de la clase Post.
     *
     * @param author    El nombre de usuario del autor
     *                  de la publicación.
     */
    public Post(String author)
    {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }

    // Métodos omitidos.

}

public class MessagePost extends Post
{
    private String message;  // Mensaje de varias líneas
                             // de longitud arbitraria.

    /**
     * Constructor para los objetos de la clase MessagePost.
     *
     * @param author    El nombre de usuario del autor
     *                  de la publicación.
     * @param text      El texto de esta publicación.
     */
    public MessagePost(String author, String text)
    {
        super(author);
        message = text;
    }

    // Métodos omitidos.
}
```

Aquí podemos hacer varias observaciones. En primer lugar, la clase `Post` tiene un constructor, aun cuando no tengamos ninguna instancia de la clase `Post` directamente.[2] Este constructor recibe los parámetros necesarios para inicializar los campos de `Post` y contiene el código para llevar a acabo esta inicialización. En segundo lugar, el constructor de `MessagePost` recibe los parámetros necesarios para inicializar tanto los campos de `Post` como de `MessagePost`. Después, incluye la siguiente línea de código:

```
super(author);
```

La palabra clave `super` es una llamada desde el constructor de la subclase al constructor de la superclase. Su efecto es que se ejecute el constructor de `Post` como parte de la ejecución del constructor de `MessagePost`. Cuando creamos una publicación de mensaje, se invoca al constructor de `MessagePost`, que a su vez invoca, como primera instrucción, al constructor de `Post`. El constructor de `Post` inicializa los campos de `Post` y luego vuelve al constructor de `MessagePost`, que inicializa el campo restante definido en la clase `MessagePost`. Para que esto funcione, los parámetros necesarios para la inicialización de los campos de la publicación se pasan al constructor de la superclase como parámetros en la llamada a `super`.

> **Concepto:**
>
> **Constructor de la superclase** El constructor de una subclase debe siempre invocar al constructor de su superclase como primera instrucción. Si el código fuente no incluye esa llamada, Java intentará insertar una llamada automáticamente.

En Java, el constructor de una subclase debe siempre invocar como primera instrucción al *constructor de la superclase*. Si no escribimos una llamada al constructor de la superclase, el compilador Java insertará automáticamente esa llamada a la superclase, con el fin de asegurarse de que los campos de la superclase se inicializan apropiadamente. La llamada insertada es equivalente a escribir:

```
super();
```

Insertar esta llamada automáticamente solo funciona si la superclase dispone de un constructor sin parámetros (porque el compilador no puede adivinar qué valores de parámetro habría que pasar). Si no es así, se informará de que se ha producido un error.

En general, es conveniente incluir siempre llamadas explícitas a la superclase en los constructores, incluso si hay alguna llamada que el compilador pudiera generar automáticamente. Incluir explícitamente las llamadas se considera un buen estilo de programación, porque evita la posibilidad de interpretaciones erróneas y de confusión en caso de que un lector no sea consciente de la generación automática de código.

Ejercicio 8.7 Establezca un punto de interrupción en la primera línea del constructor de la clase `MessagePost`. A continuación, cree un objeto `MessagePost`. Cuando aparezca la ventana del depurador, utilice *Step Into* para ejecutar el código paso a paso. Observe los campos de instancia y su inicialización. Describa sus observaciones.

[2] Actualmente, nada nos impediría crear un objeto `Post`, aunque no era esa nuestra intención cuando diseñamos estas clases. En el Capítulo 10, veremos algunas técnicas que nos permiten cerciorarnos de que no se puedan crear directamente objetos `Post`, sino solo objetos `MessagePost` o `PhotoPost`.

8.5 Adición de otros tipos de publicación a *network*

Ahora que hemos definido nuestra jerarquía de herencia para el proyecto *network* de modo que los elementos comunes de las publicaciones se encuentren en la clase `Post`, resulta mucho más fácil añadir otros tipos de publicaciones. Por ejemplo, podríamos querer añadir publicaciones de eventos, que consisten en una descripción de un evento estándar (por ejemplo, "*Fred has joined the 'Neal Stephenson fans' group.*", para indicar que Fred se ha unido al grupo de fans de Neal Stephenson). Los eventos estándar pueden ser que un usuario se una a un grupo, que un usuario se haga amigo de otro o que otro usuario cambie su imagen de perfil. Para conseguir esto, ahora podemos definir una nueva subclase de `Post` denominada `EventPost` (Figura 8.7). Dado que `EventPost` es una subclase de `Post`, hereda automáticamente todos los campos y métodos que ya hemos definido en `Post`. Por tanto, los objetos `EventPost` ya tienen un nombre de usuario, una marca temporal, un contador de votos ("Me gusta") y una lista de comentarios. A continuación, podemos concentrarnos en añadir los atributos que sean específicos de las publicaciones de eventos, como por ejemplo el tipo de evento. El tipo de evento se puede almacenar como una constante enumerada (véase el Capítulo 6) o como una cadena de caracteres que describa el evento.

Concepto:

La herencia nos permite **reutilizar** dentro de un nuevo contexto las clases previamente escritas.

Esto es un ejemplo de cómo la herencia nos permite *reutilizar* el trabajo realizado. Podemos reutilizar el código que hemos escrito para las publicaciones fotográficas y las publicaciones de mensajes (en la clase `Post`) de manera que también funcione para la clase `EventPost`. La capacidad de reutilizar los componentes software existentes es una de las grandes ventajas proporcionadas por el mecanismo de herencia. Veremos este tema con más detalle posteriormente.

Esta reutilización tiene el efecto de que hace falta mucho menos código nuevo al introducir tipos adicionales de publicación. Puesto que los nuevos tipos de publicación se pueden definir como subclases de `Post`, solo será necesario añadir el código que realmente difiera del que ya contiene `Post`.

Imagine ahora que cambiamos ligeramente los requisitos: las publicaciones de eventos en nuestra aplicación *network* no tendrá asociado un botón "Like" (Me gusta) ni tampoco una lista de comentarios. Se trata de publicaciones que solo son informativas. ¿Cómo podemos conseguir esto? Actualmente, como `EventPost` es una subclase de `Post`, hereda automáticamente los campos `likes` y `comments`. ¿Representa esto un problema?

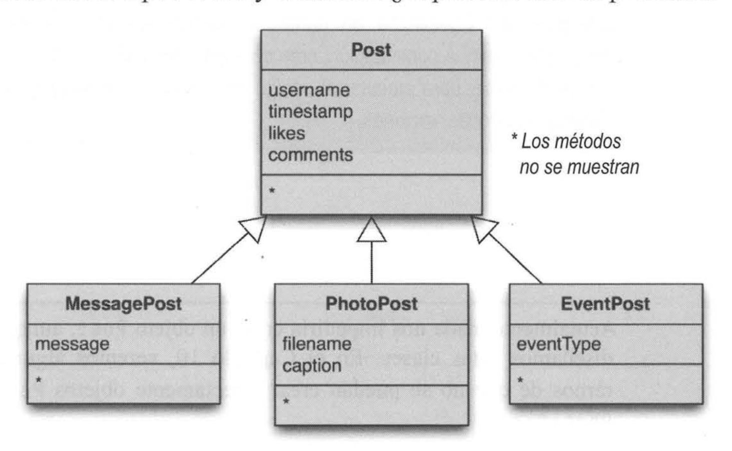


Figura 8.7
Elementos de *network* con una clase `EventPost`.

Podríamos dejar las cosas como están y decidir no mostrar nunca el contador de votos ni los comentarios para las publicaciones de eventos, nos limitaríamos a ignorar los campos. Sin embargo, esta solución no parece ser muy correcta. Tener los campos presentes pero sin utilizarlos, parece una invitación a que se produzcan problemas. Algún día, podría llegar un programador de mantenimiento que no se dé cuenta de que estos campos no deben ser utilizados y trate de procesarlos.

Alternativamente, podríamos escribir `EventPost` sin heredar de `Post`. Pero entonces volvemos al problema de la duplicación de código para los campos `username` y `timestamp` y para sus métodos.

La solución consiste en refactorizar la jerarquía de clases. Podemos introducir una nueva superclase para todas las publicaciones que tengan comentarios asociados (a la que denominaremos `CommentedPost`), que será una subclase de `Post` (Figura 8.8). Entonces pasaremos los campos `likes` y `comments` de la clase `Post` a esta nueva clase. `MessagePost` y `PhotoPost` serán ahora subclases de la nueva clase `CommentedPost`, mientras que `EventPost` heredará directamente de `Post`. Los objetos `MessagePost` heredarán todo de ambas superclases y tendrán los mismos campos y métodos que antes. Los objetos de la clase `EventPost`, por su parte, heredarán los campos `username` y `timestamp`, pero no los comentarios.

Esta es una situación muy común a la hora de diseñar jerarquías de clases. Cuando la jerarquía no parece encajar adecuadamente con el problema, tenemos que refactorizarla.

Las clases que no se pretenden utilizar para crear instancias, sino que su único propósito es servir exclusivamente como superclases de otras clases (como `Post` y `CommentedPost`), se denominan *clases abstractas*. Hablaremos de ellas con más detalle en el Capítulo 10.

Ejercicio 8.8 Abra el proyecto *network-v2*. Añada al proyecto una clase para publicaciones de eventos. Cree algunos objetos de publicación de eventos y compruebe que todos los métodos funcionan de la forma esperada.

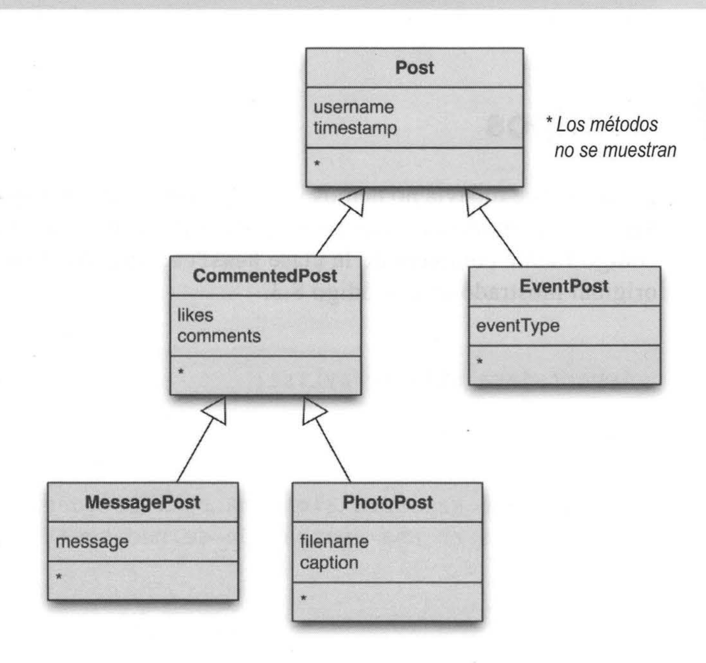


Figura 8.8
Adición de más tipos de publicaciones a *network*.

8.6 Ventajas de la herencia (hasta ahora)

Ya hemos visto varias ventajas de utilizar la herencia en la aplicación *network*. Antes de explorar otros aspectos de la herencia, vamos a resumir las ventajas generales con las que nos hemos encontrado hasta ahora:

- **Se evita la duplicación de código** El uso de la herencia nos ahorra la necesidad de escribir dos veces (o incluso más) copias idénticas o muy similares de un mismo código.
- **Reutilización del código** El código existente puede reutilizarse. Si ya existe una clase similar a la que necesitamos, podemos definir una subclase de la clase existente y reutilizar parte del código ya escrito, en lugar de tener que implementar todo de nuevo.
- **Se facilita el mantenimiento** Mantener la aplicación es más fácil, porque la relación entre las clases está claramente expresada. Un cambio en un campo o en un método compartidos entre diferentes tipos de subclases solo tendrá que hacerse una vez.
- **Ampliabilidad** Utilizando la herencia resulta mucho más fácil ampliar de determinadas maneras una aplicación existente.

Ejercicio 8.9 Ordene estos elementos en una jerarquía de herencia: manzana, helado, pan, fruta, comida, cereal, naranja, postre, mousse de chocolate, baguette.

Ejercicio 8.10 ¿En qué relación de herencia podrían estar un *panel táctil* y un *ratón*? (Estamos hablando aquí sobre dispositivos de entrada de computadora, no de esos pequeños mamíferos peludos.)

Ejercicio 8.11 En ocasiones, las cosas son más difíciles de lo que parece. Considere esta pregunta: ¿en qué tipo de relación de herencia se encuentran *Rectángulo* y *Cuadrado*? ¿Cuáles son los argumentos aplicables? Explique su respuesta.

8.7 Subtipos

Lo único que todavía no hemos investigado es cómo hay que modificar el código de la clase `NewsFeed` al modificar nuestro proyecto para utilizar la herencia. El Código 8.5 muestra el código fuente completo de la clase `NewsFeed`, que podemos comparar con el código fuente original mostrado en el Código 8.3.

Código 8.5
Código fuente de la clase `NewsFeed` (segunda versión).

```
import java.util.ArrayList;

/**
 *
 * La clase NewsFeed almacena publicaciones para la fuente de
 * noticias en una aplicación de red social (como FaceBook o
 * Google+).
 *
```

Código 8.5
(continuación)
Código fuente de la clase `NewsFeed` (segunda versión).

```
 * La visualización de las publicaciones se simula actualmente
 * imprimiendo los detalles en el terminal. (Más adelante, debería
 * mostrarse en un explorador web.)
 *
 * Esta versión no guarda los datos en disco y no proporciona
 * ninguna función de búsqueda ni de ordenación.
 *
 * @author Michael Kölling y David J. Barnes
 * @version 0.2
 */
public class NewsFeed
{
    private ArrayList<Post> posts;
    /**
     * Construye una fuente de noticias vacía.
     */
    public NewsFeed()
    {
        posts = new ArrayList<Post>();
    }

    /**
     * Añade una publicación a la fuente de noticias.
     *
     * @param post  La publicación que hay que añadir.
     */
    public void addPost(Post post)
    {
        posts.add(post);
    }

    /**
     * Mostrar la fuente de noticias. Actualmente:
     * imprimir los detalles en el terminal.
     * (Para más adelante: sustituir esto
     * por una visualización en un explorador web.)
     */
    public void show()
    {
        // Mostrar todas las publicaciones
        for(Post post : posts) {
            post.display();
            System.out.println();   // línea vacía
                                    // entre publicaciones
        }
    }
}
```

Como podemos ver, el código se ha hecho significativamente más corto y más simple, gracias a nuestro cambio consistente en modificar la herencia. Donde en la primera versión (Código 8.3) todo tenía que hacerse dos veces, ahora solo se hace una vez. Únicamente tenemos una colección, solo tenemos un método para añadir publicaciones y solo tenemos un bucle en el método `show`.

La razón por la que hemos podido acortar el código fuente es que, en la nueva versión, se puede usar el tipo `Post` en aquellos lugares en los que antes empleábamos `MessagePost` y `PhotoPost`. Investigaremos este aspecto estudiando primero el método `addPost`.

En la primera versión, teníamos dos métodos para añadir publicaciones a la fuente de noticias. Sus cabeceras eran las siguientes:

```
public void addMessagePost(MessagePost message)
public void addPhotoPost(PhotoPost photo)
```

Concepto:

Subtipo De forma análoga a la jerarquía de clases, los tipos forman una jerarquía de tipos. El tipo especificado por la definición de una subclase es un subtipo del tipo correspondiente a su superclase.

En la nueva versión, tenemos un único método para cumplir el mismo propósito:

```
public void addPost(Post post)
```

Los parámetros en la versión original están definidos con los tipos `MessagePost` y `PhotoPost`, garantizando así que pasemos objetos `MessagePost` y `PhotoPost` a esos métodos, porque los tipos de los parámetros reales deben corresponderse con los de los parámetros formales. Hasta ahora, hemos interpretado el requisito de que los tipos de parámetros se correspondan como si significara "deben ser del mismo tipo" –por ejemplo, que el nombre del tipo de un parámetro real tenga que ser el mismo que el nombre del tipo del correspondiente parámetro formal. Sin embargo, esto es solo una verdad a medias, porque podemos usar un objeto de una subclase en cualquier lugar donde lo que se requiera sea el tipo correspondiente a su superclase.

8.7.1 Subclases y subtipos

Ya hemos dicho anteriormente que las clases definen tipos. El tipo de un objeto creado a partir de la clase `MessagePost` es `MessagePost`. También hemos dicho que las clases pueden tener subclases. Por tanto, los tipos definidos por las clases pueden tener subtipos. En nuestro ejemplo, el tipo `MessagePost` es un subtipo del tipo `Post`.

8.7.2 Subtipos y asignaciones

Concepto:

Variables y subtipos Las variables pueden almacenar objetos de su tipo declarado o de cualquier subtipo de su tipo declarado.

Cuando queremos asignar un objeto a una variable, el tipo del objeto debe corresponderse con el tipo de la variable. Por ejemplo,

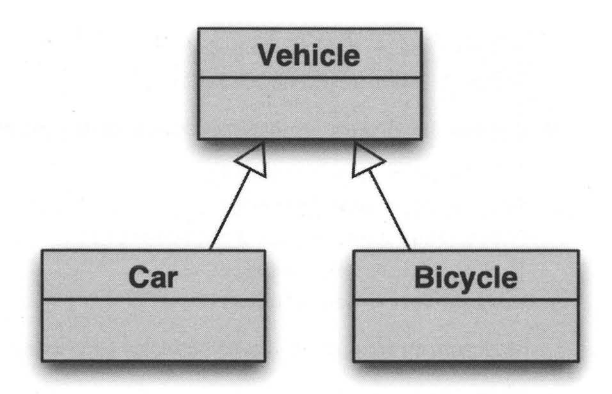
```
Car myCar = new Car();
```

es una asignación válida, porque se asigna un objeto de tipo `Car` a una variable que se ha declarado que tiene que almacenar objetos de tipo `Car`. Ahora que conocemos el concepto de herencia, es el momento de enunciar la regla de los tipos de forma más completa: una variable puede almacenar objetos de su tipo declarado o de cualquier subtipo de su tipo declarado.

Imagine que tenemos una clase `Vehicle` con dos subclases, `Car` y `Bicycle` (Figura 8.9). En este caso, la regla de los tipos indica que todas las asignaciones siguientes serían legales:

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

Figura 8.9
Una jerarquía de herencia.



El tipo de una variable declara qué es lo que esa variable puede almacenar. Declarar una variable de tipo `Vehicle` indica que esta variable puede almacenar vehículos. Pero como un coche (`Car`) es un vehículo, resulta perfectamente legal almacenar un coche en una variable pensada para vehículos. (Piense en la variable como si fuera un garaje: si alguien nos dice que podemos aparcar un vehículo en un garaje, pensaríamos que está permitido aparcar tanto un coche como una bicicleta.)

Concepto:

Sustitución Pueden utilizarse objetos de un subtipo en cualquier lugar en el que lo que se espere sean objetos de un supertipo. Esta posibilidad se conoce con el nombre de sustitución.

Este principio se conoce como *sustitución.* En los lenguajes orientados a objetos, cuando se espera un objeto de una superclase podemos sustituirlo por un objeto de una de sus subclases, porque el objeto de la subclase es un caso especial de la superclase. Por ejemplo, si alguien nos pide que compremos un perro, podemos satisfacer la petición perfectamente comprando un caniche o un dálmata. Tanto el caniche como el dálmata son subclases de perro, por lo que suministrar cualquiera de los dos cuando lo que se espera es un objeto de la clase `Perro` es perfectamente admisible.

Sin embargo, lo que no se puede es hacer lo contrario:

```
Car c1 = new Vehicle(); // ¡Esto es un error!
```

Esta instrucción trata de almacenar un objeto `Vehicle` en una variable `Car`. Java no lo permitirá y nos informará de un error si intentamos compilar esta instrucción. La variable está declarada para almacenar coches y un vehículo, por su parte, puede ser o no un coche –no lo sabemos. Por tanto, la instrucción puede ser incorrecta y no se permite.

De forma similar:

```
Car c2 = new Bicycle(); // ¡Esto es un error!
```

Esta también es una instrucción ilegal. Una bicicleta no es un coche (o más formalmente, el tipo `Bicycle` no es un subtipo de `Car`), por lo que la asignación no está permitida.

Ejercicio 8.12 Suponga que tenemos cuatro clases: `Person`, `Teacher`, `Student` y `PhDStudent`. `Teacher` y `Student` son ambas subclases de `Person`. `PhDStudent` es una subclase de `Student`.

a. ¿Cuáles de las siguientes asignaciones son legales? ¿Por qué?

```
Person p1 = new Student();
Person p2 = new PhDStudent();
PhDStudent phd1 = new Student();
```

```
Teacher t1 = new Person();
Student s1 = new PhDStudent();
```

b. Suponga que tenemos las siguientes declaraciones y asignaciones legales:

```
Person p1 = new Person();
Person p2 = new Person();
PhDStudent phd1 = new PhDStudent();
Teacher t1 = new Teacher();
Student s1 = new Student();
```

Basándose en ellas, ¿cuáles de las siguientes asignaciones serían legales? ¿Por qué?

```
s1 = p1;
s1 = p2;
p1 = s1;
t1 = s1;
s1 = phd1;
phd1 = s1;
```

Ejercicio 8.13 Compruebe sus respuestas a la pregunta anterior creando versiones básicas de las clases mencionadas en ese ejercicio y probándolas en BlueJ.

8.7.3 Subtipos y paso de parámetros

Pasar un parámetro (es decir, asignar un parámetro real a la variable correspondiente a un parámetro formal) tiene exactamente el mismo comportamiento que una asignación a una variable. Esta es la razón por la que podemos pasar un objeto de tipo `MessagePost` a un método que tiene un parámetro de tipo `Post`. Tenemos la siguiente definición del método `addPost` en la clase `NewsFeed`:

```
public class NewsFeed
{
    public void addPost(Post post)
    {
        . . .
    }
}
```

Ahora podemos utilizar este método para añadir publicaciones de mensajes y publicaciones fotográficas a la fuente de noticias:

```
NewsFeed feed = new NewsFeed();
MessagePost message = new MessagePost(...);
PhotoPost photo = new PhotoPost(...);

feed.addPost(message);
feed.addPost(photo);
```

Gracias a las reglas de los subtipos, solo necesitamos un método (con un parámetro de tipo `Post`) para añadir tanto objetos `MessagePost` como objetos `PhotoPost`.

Hablaremos de los subtipos con más detalle en el siguiente capítulo.

8.7.4 Variables polimórficas

Las variables que almacenan tipos de objetos en Java son variables *polimórficas*. El término "polimórfico" (que significa *multiforme*) hace referencia al hecho de que una variable puede almacenar objetos de diferentes tipos (en concreto, del tipo declarado o de cualquier subtipo del tipo declarado). El polimorfismo aparece en los lenguajes orientados a objetos en distintos contextos –las variables polimórficas son solo el primer ejemplo. Veremos otros aspectos del polimorfismo con mayor detalle en el siguiente capítulo.

Por el momento, vamos a limitarnos a observar cómo el uso de variables polimórficas nos ayuda a simplificar nuestro método `show`. El cuerpo de este método es el siguiente:

```
for(Post post : posts) {
    post.display();
    System.out.println();     // línea vacía entre publicaciones.
}
```

Aquí, iteramos a través de la lista de publicaciones (almacenada en un `ArrayList` en la variable `posts`). Extraemos cada publicación y luego invocamos su método `display`. Observe que las publicaciones que extraemos de la lista son de tipo `MessagePost` o `PhotoPost`, no de tipo `Post`. Sin embargo, podemos utilizar una variable de bucle de tipo `Post`, porque las variables son polimórficas. La variable `post` es capaz de contener objetos `MessagePost` y `PhotoPost`, porque ambos son subtipos de `Post`.

Por tanto, el uso de la herencia en este ejemplo ha eliminado la necesidad de disponer de dos bucles separados en el método `show`. La herencia evita la duplicación de código no solo en las clases servidoras, sino también en las clases cliente de dichas clases.

Nota Al hacer los ejercicios puede haber observado que el método `show` tiene un problema: no se imprimen todos los detalles. Resolver este problema requiere algunas explicaciones adicionales, que proporcionaremos en el siguiente capítulo.

Ejercicio 8.14 ¿Qué habría que cambiar en la clase `NewsFeed` si añadiéramos otra subclase de `Post` (por ejemplo, una clase `EventPost`)? ¿Por qué?

8.7.5 *Casting* o proyección de tipos

En ocasiones, la regla de que no podemos asignar de un supertipo a un subtipo es más restrictiva de lo necesario. Si sabemos que la variable correspondiente al supertipo almacena un objeto del subtipo, podría permitirse sin problemas la asignación. Por ejemplo:

```
Vehicle v;
Car c = new Car();
v = c; // correcto
c = v; // error
```

Las instrucciones anteriores no podrían compilarse. Obtendríamos un error del compilador en la última línea, porque no está permitido asignar una variable de tipo `Vehicle` a una variable de tipo `Car` (supertipo a subtipo). Sin embargo, si ejecutamos estas instrucciones por orden

sabemos que esta asignación podría perfectamente autorizarse. Podemos ver que la variable `v` contiene en realidad un objeto de tipo `Car`, por lo que asignarla a `c` sería correcto, pero el compilador no es tan inteligente: traduce el código línea por línea, de modo que examina la última línea de manera aislada sin saber qué es lo que hay realmente almacenado en la variable `v`. Este fenómeno se denomina pérdida de tipos. El tipo del objeto almacenado en `v` es realmente `Car`, pero el compilador no lo sabe.

Podemos resolver este problema informando explícitamente al sistema de tipos de que la variable `v` almacena un objeto `Car`. Para hacer esto, tenemos que emplear un *operador de cast* u *operador de proyección de tipos*:

```
c = (Car) v; // correcto
```

El operador de cast está compuesto por el nombre de un tipo (aquí, `Car`) escrito entre paréntesis delante de una variable o una expresión. Al hacer esto, se hace que el compilador crea que el objeto es un `Car`, por lo que no generará un error. Sin embargo, en tiempo de ejecución, el sistema Java comprobará que se trata realmente de un `Car`. Si hemos tenido cuidado y verdaderamente es un objeto `Car`, no habrá problemas. Pero si el objeto almacenado en `v` es de otro tipo, el sistema de tiempo de ejecución indicará que hay un error (denominado `ClassCastException`) y el programa se detendrá.[3]

Considere ahora el siguiente fragmento de código en el que `Bicycle` es también una subclase de `Vehicle`:

```
Vehicle v;
Car c;
Bicycle b;
c = new Car();
v = c; // correcto
b = (Bicycle) c; // ¡error en tiempo de compilación!
b = (Bicycle) v; // ¡error en tiempo de ejecución!
```

Las dos últimas asignaciones fallarán. El intento de asignar `c` a `b` (incluso con el cast) nos dará un error en tiempo de compilación. El compilador se da cuenta de que `Car` y `Bicycle` no forman una relación subtipo/supertipo, por lo que `c` nunca puede almacenar un objeto `Bicycle` –la asignación no podría funcionar nunca.

El intento de asignar `v` a `b` (con el cast) será aceptado en tiempo de compilación, pero fallará en tiempo de ejecución. `Vehicle` es una superclase de `Bicycle`, por lo que `v` podría potencialmente contener un objeto `Bicycle`. Sin embargo, en tiempo de ejecución, resulta que el objeto almacenado en `v` no es de tipo `Bicycle` sino de tipo `Car`, por lo que el programa terminará prematuramente.

El *casting* debe evitarse siempre que sea posible, porque puede conducir a errores en tiempo de ejecución, y evidentemente eso es algo que no queremos. El compilador no puede ayudarnos a garantizar la corrección del programa en este caso.

En la práctica, el *casting* rara vez se necesita en un programa orientado a objetos bien estructurado. En casi todos los casos, cuando se utiliza un cast en el código, podría reestructurarse el código para evitar ese cast y terminar disponiendo de un programa mejor diseñado. Esto

---

[3] Las excepciones se explican en detalle en el Capítulo 12.

implica normalmente sustituir el cast por una llamada a un método polimórfico (hablaremos más sobre este tema en el capítulo siguiente).

8.8 La clase `Object`

Todas las clases tiene una superclase. Hasta ahora, podría parecer que la mayoría de las clases que hemos visto no tienen una superclase, pero de hecho, aunque podemos declarar una superclase explícita para cada clase, todas las clases que no tienen ninguna declaración explícita de superclase heredan implícitamente de una clase denominada `Object`.

Concepto:

Todas las clases que no tienen una superclase explícita tienen a `Object` como superclase.

`Object` es una clase de la librería estándar Java que sirve como superclase para todos los objetos. Escribir una declaración de clase como

```
public class Person
{
    ...
}
```

es equivalente a escribir

```
public class Person extends Object
{
    ...
}
```

El compilador Java inserta automáticamente la superclase `Object` para todas las clases que no tengan una declaración `extends` explícita, así que no necesitamos hacerlo nosotros. Todas las clases (con la única excepción de la propia clase `Object`) heredan de `Object`, directa o indirectamente. La Figura 8.10 muestra algunas clases seleccionadas aleatoriamente para ilustrar este punto.

Disponer de una superclase común para todos los objetos tiene dos objetivos. En primer lugar, podemos declarar variables polimórficas de tipo `Object` capaces de almacenar cualquier tipo

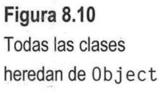
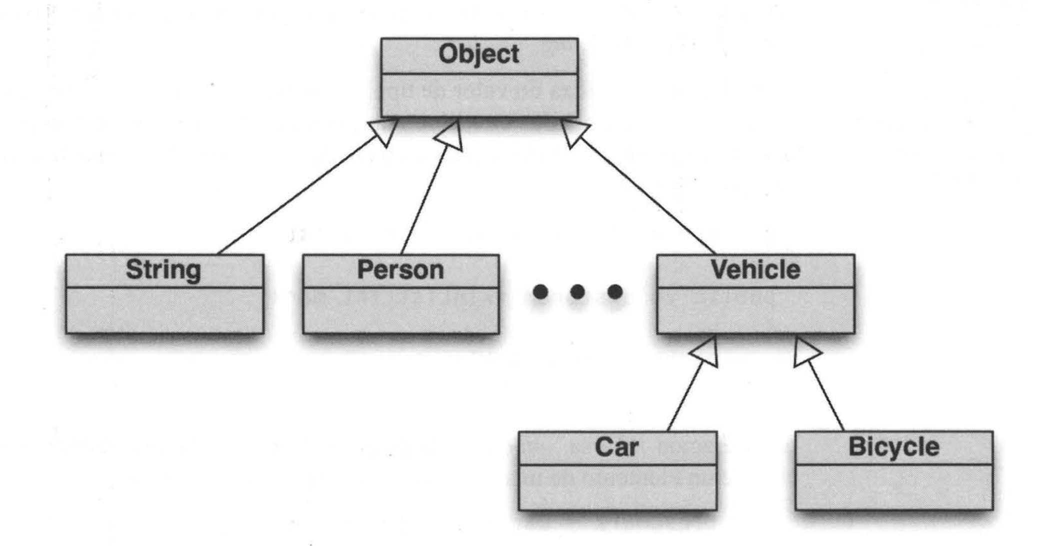


Figura 8.10 Todas las clases heredan de `Object`



de objeto. Disponer de variables que puedan almacenar cualquier tipo de objeto no solo resulta útil muy a menudo, sino que también puede ser de gran ayuda en algunas situaciones. En segundo lugar, la clase `Object` puede definir algunos métodos que luego estén disponibles automáticamente para todos los objetos existentes. De especial importancia son los métodos `toString`, `equals` y `hashCode` que `Object` define. Este segundo punto tendrá un gran interés más adelante, y lo discutiremos con mayor detalle en el siguiente capítulo.

8.9 *Autoboxing* y clases envoltorio

Ya hemos visto que, con una parametrización adecuada, las clases de colección pueden almacenar objetos de cualquier tipo de objeto. Pero sigue existiendo un problema: Java tiene algunos tipos que no son tipos de objeto.

Como sabemos, los tipos simples, como `int`, `boolean` y `char`, están separados de los tipos de objeto. Sus valores no son instancias de clase y no heredan de la clase `Object`. Debido a esto, no son subtipos de `Object` y, normalmente, no sería posible añadirlos a una colección.

Esto es algo bastante inconveniente, ya que hay situaciones en las que nos gustaría, por ejemplo, crear una lista de valores `int` o de valores `char`. ¿Qué es lo que podemos hacer?

La solución que Java da a este problema son las *clases envoltorio.* Cada tipo primitivo de Java tiene una clase envoltorio correspondiente que representa el mismo tipo, pero que es un tipo de objeto real. La clase envoltorio para `int`, por ejemplo, se denomina `Integer`. En el Apéndice B se proporciona una lista completa de los tipos simples y de sus correspondientes clases envoltorio.

La siguiente instrucción envuelve explícitamente en un objeto `Integer` el valor de la variable `ix`, que es del tipo primitivo `int`:

```
Integer iwrap = new Integer(ix);
```

Concepto:

El ***autoboxing*** se lleva a cabo automáticamente cada vez que se utiliza un valor de tipo primitivo dentro de un contexto en el que se requiere un tipo envoltorio.

Y ahora `iwrap` podría, obviamente, almacenarse por ejemplo con toda sencillez en una colección `ArrayList<Integer>`. Sin embargo, el almacenar valores primitivos en una colección de objetos es todavía más fácil gracias a una característica del compilador denominada *autoboxing.*

Cada vez que se utiliza un valor de tipo primitivo en un contexto en el que se requiere un tipo envoltorio, el compilador envuelve automáticamente el valor de tipo primitivo en un objeto envoltorio apropiado. Esto significa que podemos añadir directamente a una colección valores de tipo primitivo:

```
private ArrayList<Integer> markList;
. . .
public void storeMarkInList(int mark)
{
    markList.add(mark);
}
```

La operación inversa, *unboxing*, también se realiza automáticamente, por lo que una extracción de un elemento de una colección podría tener el siguiente aspecto:

```
int firstMark = markList.remove(0);
```

El *autoboxing* también se aplica siempre que se pasa un valor de tipo primitivo como parámetro a un método que espera un tipo envoltorio, así como cuando se almacena un valor de tipo primitivo en una variable de tipo envoltorio. De forma similar, la operación de *unboxing* se aplica cuando se pasa un valor de tipo envoltorio como parámetro a un método que espera un valor de tipo primitivo, así como cuando se almacena un valor de tipo envoltorio en una variable de tipo primitivo. Merece la pena resaltar que, aunque parezca con esto que los tipos primitivos pueden almacenarse en colecciones, en realidad el tipo de la colección debe continuar declarándose mediante el tipo envoltorio (por ejemplo, `ArrayList<Integer>` y no `ArrayList<int>`).

8.10 La jerarquía de colecciones

La librería Java utiliza intensivamente la herencia a la hora de definir las clases de colecciones. Por ejemplo, la clase `ArrayList` hereda de una clase denominada `AbstractList`, que a su vez hereda de `AbstractCollection`. No vamos a analizar esta jerarquía, porque está descrita en detalle en varios lugares fácilmente accesibles. Puede encontrar una buena descripción en el sitio web de Oracle en `http://download.oracle.com/javase/tutorial/collections/index.html`.

Observe que algunos detalles de esta jerarquía requieren comprender las *interfaces* Java. Hablaremos de ellas en el Capítulo 10.

Ejercicio 8.15 Utilice la documentación de las librerías de clases estándar de Java para localizar información sobre la jerarquía de herencia de las clases de colecciones. Dibuje un diagrama que muestre dicha jerarquía.

8.11 Resumen

En este capítulo hemos presentado una primera visión del concepto de herencia. Todas las clases de Java están organizadas en una jerarquía de herencia. Cada clase puede tener una superclase declarada explícitamente, o bien heredar implícitamente de la clase `Object`.

Las subclases suelen representar especializaciones de las superclases. Debido a ello, la relación de herencia también suele denominarse en ocasiones relación de tipo es-un (un coche *es-un* vehículo).

Las subclases heredan todos los campos y métodos de su superclase. Los objetos de las subclases disponen de todos los campos y métodos declarados en su propia clase, así como de los de todas sus superclases. Las relaciones de herencia se pueden utilizar para evitar la duplicación de código, reutilizar el código existente y para hacer que una aplicación sea más mantenible y ampliable.

Las subclases forman subtipos, lo que conduce a la existencia de variables polimórficas. Los objetos del supertipo pueden sustituirse por objetos del subtipo, y las variables pueden almacenar objetos que sean instancias de subtipos de su tipo declarado.

La herencia permite diseñar estructuras de clase que son más fáciles de mantener y más flexibles. Este capítulo proporciona únicamente una introducción al uso de la herencia con el

propósito de mejorar las estructuras de los programas. En los siguientes capítulos analizaremos otros usos de la herencia, así como sus ventajas.

Términos introducidos en el capítulo

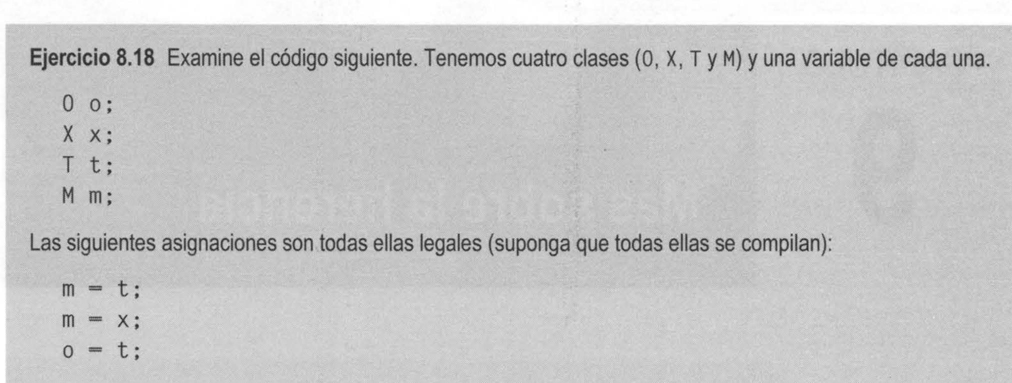
herencia, superclase (padre), subclase (hija), es-un, jerarquía de herencia, clase abstracta, sustitución de subtipos, variable polimórfica, pérdida de tipos, *cast* (proyección), *autoboxing*, clases envoltorio

Resumen de conceptos

- **herencia** La herencia nos permite definir una clase como ampliación de otra.
- **superclase** Llamamos superclase a toda clase que es ampliada por otra clase.
- **subclase** Llamamos subclase a una clase que amplía (hereda de) otra clase. La subclase hereda todos los campos y métodos de su superclase.
- **jerarquía de herencia** Las clases que están vinculadas por relaciones de herencia forman una jerarquía de herencia.
- **constructor de la superclase** El constructor de una subclase debe siempre invocar al constructor de su superclase como primera instrucción. Si el código fuente no incluye esa llamada, Java intentará insertar una llamada automáticamente.
- **reutilización** La herencia nos permite reutilizar dentro de un nuevo contexto las clases previamente escritas.
- **subtipo** De forma análoga a la jerarquía de clases, los tipos forman una jerarquía de tipos. El tipo especificado por la definición de una subclase es un subtipo del tipo correspondiente a su superclase.
- **variables y subtipos** Las variables pueden almacenar objetos de su tipo declarado o de cualquier subtipo de su tipo declarado.
- **sustitución** Pueden utilizarse objetos de un subtipo en cualquier lugar en el que lo que se espere sean objetos de un supertipo. Esta posibilidad se conoce con el nombre de sustitución.
- **Object** Todas las clases que no tienen una superclase explícita tienen a `Object` como superclase.
- ***autoboxing*** El *autoboxing* se lleva a cabo automáticamente cada vez que se utiliza un valor de tipo primitivo dentro de un contexto en el que se requiere un tipo envoltorio.

Ejercicio 8.16 Vuelva al proyecto *lab-classes* del Capítulo 1. Añada profesores al proyecto (cada clase de laboratorio puede tener varios estudiantes y un único profesor). Utilice la herencia para evitar duplicar el código entre los estudiantes y los profesores (ambos tienen un nombre, unos datos de contacto, etc.).

Ejercicio 8.17 Dibuje una jerarquía de herencia que represente las partes de un sistema de computadora (procesador, memoria, unidad de disco duro, unidad de DVD, impresora, escáner, teclado, ratón, etc.).



Ejercicio 8.18 Examine el código siguiente. Tenemos cuatro clases (`O`, `X`, `T` y `M`) y una variable de cada una.

```
O o;
X x;
T t;
M m;
```

Las siguientes asignaciones son todas ellas legales (suponga que todas ellas se compilan):

```
m = t;
m = x;
o = t;
```

Las siguientes asignaciones son todas ellas ilegales (producen errores de compilación):

```
o = m;
o = x;
x = o;
```

¿Qué podemos deducir acerca de las relaciones de estas clases? Dibuje un diagrama de clases.

Ejercicio 8.19 Dibuje una jerarquía de herencia de `AbstractList` y de todas sus subclases (directas e indirectas), tal como están definidas en la librería estándar Java.

CAPÍTULO

9 Más sobre la herencia

Principales conceptos explicados en el capítulo:

- polimorfismo de métodos
- sustitución de métodos
- tipo estático y dinámico
- búsqueda dinámica de métodos

Estructuras Java explicadas en este capítulo:

`super` (en métodos), `toString`, `protected`, `instanceof`

En el último capítulo hemos introducido los conceptos principales de la herencia analizando el ejemplo *network*. Aunque hemos visto los fundamentos básicos de la herencia, siguen existiendo numerosos detalles de importancia que aun no hemos investigado. La herencia es crucial para comprender y utilizar los lenguajes orientados a objetos, y es necesario comprender sus detalles para poder seguir avanzando.

En este capítulo, continuaremos utilizando el ejemplo *network* para explorar lo más importante de las cuestiones restantes en relación con la herencia y el polimorfismo.

9.1 El problema: el método de visualización en *network*

Al experimentar con los ejemplos de *network* en el Capítulo 8, probablemente habrá observado que la segunda versión (la que utiliza la herencia) tiene un problema: el método `display` no muestra todos los datos de una publicación.

Veamos un ejemplo. Suponga que creamos un objeto `MessagePost` y otro `PhotoPost` con los siguientes datos:

La publicación de mensaje:

```
Leonardo da Vinci

Had a great idea this morning.
But now I forgot what it was. Something to do with flying ...

40 seconds ago. 2 people like this.

No comments.
```

La publicación fotográfica:

Alexander Graham Bell

[experiment.jpg] I think I might call this thing 'telephone'.

12 minutes ago. 4 people like this.

No comments.

Si introducimos estos objetos en la fuente de noticias[1] y después invocamos la primera versión del método `show` de la fuente de noticias (la que no utiliza herencia), se imprimirá

```
Leonardo da Vinci
Had a great idea this morning.
But now I forgot what it was. Something to do with flying ...
40 seconds ago  -  2 people like this.
  No comments.

Alexander Graham Bell
[experiment.jpg]
I think I might call this thing 'telephone'.
12 minutes ago  -  4 people like this.
  No comments.
```

Aunque el formato no es muy adecuado (porque en el terminal de texto no tenemos opciones para dar formato), toda la información está ahí, y podemos imaginarnos cómo podría adaptarse posteriormente el método `show` para mostrar los datos con un formato más elegante en una interfaz de usuario distinta.

Compare esto con la segunda versión de *network* (con herencia), que solo imprime

```
Leonardo da Vinci
40 seconds ago  -  2 people like this.
  No comments.

Alexander Graham Bell
12 minutes ago  -  4 people like this.
  No comments.
```

Observamos que el texto de la publicación de mensaje, así como el nombre del archivo de imagen y el título de la publicación fotográfica, no aparecen. La razón es muy simple, el método `display` en esta versión está implementado en la clase `Post`, no en `MessagePost` y `PhotoPost` (Figura 9.1). En los métodos de `Post`, solo están disponibles los campos declarados en `Post`. Si tratáramos de acceder al campo `message` de `MessagePost` desde el método `display` de `Post`, el sistema nos daría un error. Esto ilustra el importante principio de que la herencia es un camino unidireccional: `MessagePost` hereda los campos de `Post`, pero `Post` sigue sin conocer ningún detalle acerca de los campos de sus subclases.

[1] El texto para la publicación de mensaje es una cadena de caracteres de dos líneas. Podemos introducir un texto de varias líneas en una cadena utilizando "\n" en la cadena como salto de línea.

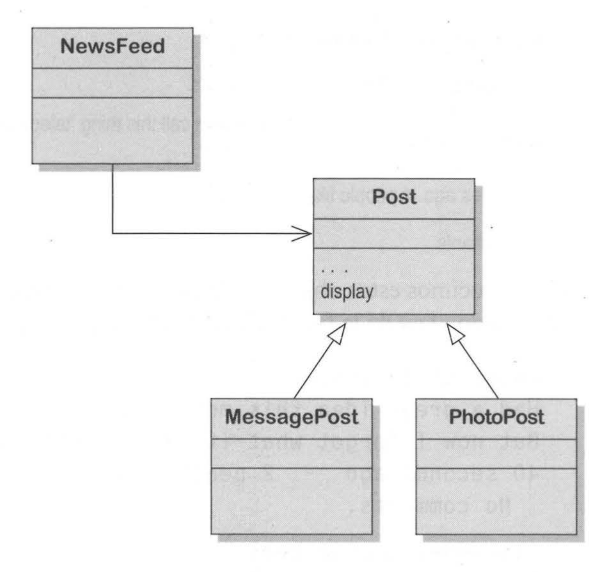


Figura 9.1
Visualización, versión 1: método `display` en la superclase.

9.2 Tipo estático y tipo dinámico

Tratar de resolver el problema de desarrollar un método `display` completamente polimórfico nos lleva a intentar explicar los *tipos estáticos* y los *tipos dinámicos* y el mecanismo de *búsqueda de métodos*. Pero comencemos por el principio.

Un primer intento de resolver el problema de la visualización podría ser mover el método `display` a las subclases (Figura 9.2). De esta forma, como el método pertenece ahora a las clases `MessagePost` y `PhotoPost`, sí que podría acceder a los campos específicos de `MessagePost` y `PhotoPost`. También podría acceder a los campos heredados, llamando a los métodos selectores definidos en la clase `Post`. Eso permitiría a los dos métodos mostrar de nuevo un conjunto completo de información. Trate de implementar esta solución completando el Ejercicio 9.1.

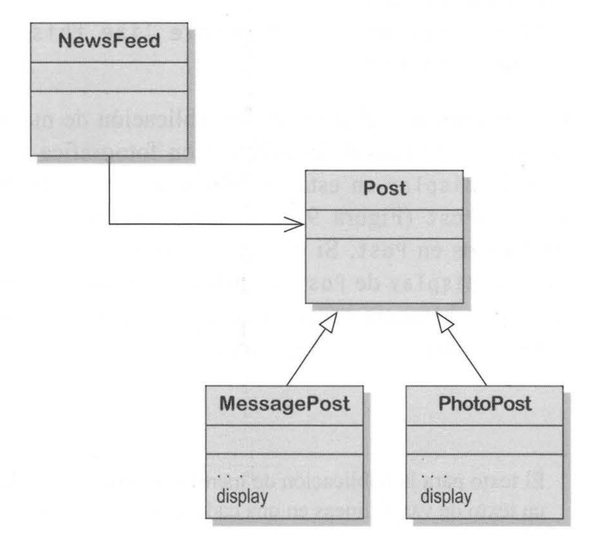


Figura 9.2
Visualización, versión 2: método `display` en las subclases.

Ejercicio 9.1 Abra su última versión del proyecto *network*. (Puede usar *network-v2* si todavía no tiene su propia versión.) Elimine el método `display` de la clase `Post` y muévalo a las clases `MessagePost` y `PhotoPost`. Compile. ¿Qué es lo que observa?

Cuando tratamos de mover el método `display` de `Post` a las subclases, observamos que se presenta algún problema: el proyecto ya no se compila. Hay dos problemas fundamentales:

- Obtenemos errores en las clases `MessagePost` y `PhotoPost`, porque no podemos acceder a los campos de la superclase.
- Obtenemos un error en la clase `NewsFeed`, porque no puede encontrar el método `display`.

La razón del primero de esos dos tipos de errores es que los campos de `Post` tienen acceso privado, por lo que son inaccesibles por parte de las otras clases, incluyendo las subclases. Puesto que no queremos romper la encapsulación y hacer estos campos públicos, la forma más fácil de resolver este problema, como se sugirió anteriormente, consiste en definir métodos selectores públicos para esos campos. Sin embargo, en la Sección 9.9, introduciremos un tipo de adicional de acceso diseñado específicamente para soportar la relación superclase–subclase.

La razón del segundo tipo error requiere una explicación más detallada, por lo que vamos a dedicarle la siguiente sección.

9.2.1 Invocación de `display` desde `NewsFeed`

En primer lugar, investigaremos el problema de invocar el método `display` desde `NewsFeed`. Las líneas relevantes de código en la clase `NewsFeed` son:

```
for(Post post : posts) {
    post.display();
    System.out.println();
}
```

La instrucción for-each extrae cada publicación de la colección; la primera instrucción del cuerpo trata de invocar el método `display` para esa publicación. El compilador nos informa de que no puede encontrar un método `display` para la publicación.

Por un lado, parece lógico: `Post` ya no tiene ningún método `display` (véase la Figura 9.2).

Por otro lado, tiene también algo de ilógico y desconcertante. Sabemos que cada objeto `Post` de la colección es, de hecho, un objeto `MessagePost` o `PhotoPost`, y ambos tienen métodos `display`. Esto debería implicar que `post.display()` funcionara correctamente, porque independientemente de lo que sea esa publicación (`MessagePost` o `PhotoPost`), sabemos que dispone de un método `display`.

Para entender en detalle por qué no funciona, necesitamos examinar más de cerca los tipos. Considere la siguiente instrucción:

```
Car c1 = new Car();
```

Decimos que el tipo de `c1` es `Car`. Antes de toparnos con la herencia, no había ninguna necesidad de distinguir si la frase "tipo de `c1`" significa "el tipo de la variable `c1`" o "el tipo del objeto almacenado en `c1`". No importaba, porque el tipo de la variable y el tipo del objeto coincidían siempre.

Pero ahora que sabemos de la existencia de subtipos tenemos que ser más precisos. Considere la siguiente instrucción:

```
Vehicle v1 = new Car();
```

¿Cuál es el tipo de v1? Eso depende precisamente de qué queramos decir con la frase "tipo de v1". El tipo de la variable v1 es Vehicle; el tipo del objeto almacenado en v1 es Car. A través de los subtipos y de las reglas de sustitución, ahora aparecen situaciones en las que el tipo de la variable y el tipo del objeto almacenado en la misma no coinciden exactamente.

Concepto:

El **tipo estático** de una variable v es el tipo tal como está declarado en el código fuente, en la instrucción de declaración de la variable.

Concepto

El **tipo dinámico** de una variable v es el tipo del objeto que está almacenado actualmente en v.

Introduzcamos una cierta terminología para que resulte más fácil explicar esta cuestión:

- Al tipo declarado de la variable lo denominamos *tipo estático,* porque está declarado en el código fuente –la representación estática del programa.
- Al tipo del objeto almacenado en una variable lo denominamos *tipo dinámico,* porque depende de las asignaciones en tiempo de ejecución –es decir, del comportamiento dinámico del programa.

Por tanto, aplicando las explicaciones anteriores, podemos ser más precisos: el tipo estático de v1 es Vehicle, mientras que el tipo dinámico de v1 es Car. Ahora podemos reformular nuestro análisis de la llamada al método display de la publicación dentro de la clase NewsFeed. En el momento de la llamada

```
post.display();
```

el tipo estático de post es Post, mientras que el tipo dinámico es MessagePost o PhotoPost (Figura 9.3). No sabemos cuál de estos dos es, suponiendo que hayamos introducido tanto objetos MessagePost como PhotoPost en la fuente de noticias.



Figura 9.3
Variable de tipo Post que contiene un objeto de tipo PhotoPost.

El compilador nos da un error porque para la comprobación de tipos se utiliza el tipo estático. El tipo dinámico a menudo solo se conoce en tiempo de ejecución, por lo que el compilador no tiene ninguna opción salvo usar el tipo estático si es que desea hacer alguna comprobación en tiempo de compilación. El tipo estático de post es Post, y Post no tiene un método display. A este respecto no tiene ninguna importancia que todos los subtipos conocidos de Post dispongan de un método display. El comportamiento del compilador es razonable a este respecto, porque no tiene ninguna garantía de que *todas* las subclases de Post vayan, ciertamente, a definir un método display y esto es imposible de comprobar en la práctica.

En otras palabras, para que esto funcione la clase Post tiene que disponer de un método display, así que parece que estamos otra vez en nuestro problema original sin haber hecho ningún progreso.

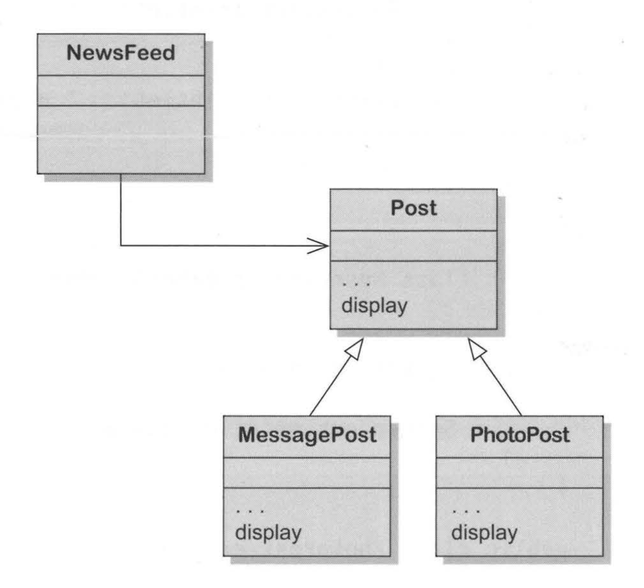
Ejercicio 9.2 En su proyecto *network*, añada otra vez un método `display` a la clase `Post`. Por ahora, escriba el cuerpo del método con una única instrucción que se limite a imprimir el nombre de usuario. A continuación, modifique los métodos `display` de `MessagePost` y `PhotoPost` de modo que la versión de `MessagePost` imprima solo el mensaje y la versión de `PhotoPost` imprima solo el título. Esto hará que se eliminen los otros errores con los que nos hemos topado anteriormente (hablaremos sobre ellos más adelante).

Ahora debería tener una situación como la que se ilustra en la Figura 9.4, con sendos métodos `display` en las tres clases. Compile su proyecto. (Si hay errores, elimínelos. Este diseño debería funcionar.)

Antes de ejecutar la aplicación, trate de predecir cuál de los métodos `display` será invocado cuando se ejecute el método `show` de la fuente de noticias.

Compruébelo. Introduzca una publicación de mensaje y una publicación fotográfica en la fuente de noticias e invoque el método `show` de la fuente de noticias. ¿Qué métodos `display` se han ejecutado? ¿Era correcta su predicción? Explique sus observaciones.

Figura 9.4
Visualización, versión 3: método `display` en las subclases y en la superclase.



9.3 Sustitución de métodos

El siguiente diseño que presentaremos es uno en el que tanto la superclase como las subclases disponen de un método `display` (Figura 9.4). La cabecera de todos los métodos `display` es exactamente la misma.

El Código 9.1 muestra los detalles relevantes del código fuente de las tres clases. La clase `Post` tiene un método `display` que imprime todos los campos declarados en `Post` (los que son comunes a las publicaciones de mensajes y a las publicaciones fotográficas), mientras que las subclases `MessagePost` y `PhotoPost` imprimen los campos específicos de los objetos `MessagePost` y `PhotoPost`, respectivamente.

Código 9.1
Código fuente de los métodos `display` de las tres clases.

```
public class Post
{
    . . .
    public void display()
    {
        System.out.println(username);
        System.out.print(timeString(timestamp));

        if(likes > 0) {
            System.out.println("  -  " + likes + " people like
                                this.");
        }
        else {
            System.out.println();
        }

        if(comments.isEmpty()) {
            System.out.println("   No comments.");
        }
        else {
            System.out.println("   " + comments.size() +
                               " comment(s). Click here to view.");
        }
    }
}

public class MessagePost extends Post
{
    . . .
    public void display()
    {
        System.out.println(message);
    }
}

public class PhotoPost extends Post
{
    . . .
    public void display()
    {
        System.out.println("  [" + filename + "]");
        System.out.println("  " + caption);
    }
}
```

Concepto:

Sustitución de métodos
Una subclase puede sustituir la implementación de un método. Para ello, la subclase declara un método con la misma signatura que la superclase, pero con un cuerpo del método diferente. El método sustituto tiene precedencia en las llamadas a método efectuadas sobre los objetos de la subclase.

Este diseño funciona algo mejor. Se compila y puede ejecutarse (aun cuando todavía no es perfecto). En el proyecto *network-v3* se proporciona una implementación de este diseño. (Si ha hecho el Ejercicio 9.2 ya dispondrá de una implementación similar de este diseño en su propia versión.)

La técnica que estamos utilizando aquí se denomina *sustitución de métodos* (en algunas ocasiones también se denomina *redefinición*). La sustitución de métodos es una situación en la que un método se define en una superclase (en este ejemplo, el método `display` en la clase `Post`), y en la subclase se define otro método con exactamente la misma signatura.

En esta situación, los objetos de la subclase dispondrán de dos métodos con el mismo nombre y la misma cabecera: uno heredado de la superclase y otro de la subclase. ¿Cuál de ellos se ejecutará cuando invoquemos este método?

9.4 Búsqueda dinámica de métodos

Un detalle sorprendente es lo que se imprime al ejecutar el método `show` de la fuente de noticias. Si volvemos a crear e introducir los objetos descritos en la Sección 9.1, la salida del método `show` en nuestra nueva versión del programa será:

```
Had a great idea this morning.
But now I forgot what it was. Something to do with flying ...

  [experiment.jpg]
  I think I might call this thing 'telephone'.
```

Podemos ver, a partir de esta salida, que lo que se han ejecutado son los métodos `display` de `MessagePost` y `PhotoPost`, no el de `Post`.

Esto puede parecer extraño a primera vista. Nuestras investigaciones de la Sección 9.2 mostraron que el compilador insistía en que hubiera un método `display` en la clase `Post` –no era suficiente con disponer de métodos en las subclases. Este experimento muestra ahora que el método de la clase `Post` no se llega a ejecutar en absoluto, sino que lo que se ejecuta son los métodos de la subclase. En resumen:

- La comprobación de tipos emplea el tipo estático, pero en tiempo de ejecución, se ejecutan los métodos correspondientes al tipo dinámico.

Este es un hecho muy importante. Para entenderlo mejor, examinemos en detalle el modo en que se invocan los métodos. Este procedimiento se conoce con el nombre de *búsqueda de métodos*, *asociación de métodos* o *despacho de métodos*. En este libro utilizaremos el término "búsqueda de métodos".

Comenzaremos con un escenario simple de búsqueda de métodos. Suponga que tenemos un objeto de la clase `PhotoPost` almacenado en una variable `v1` cuyo tipo declarado es `PhotoPost` (Figura 9.5). La clase `PhotoPost` tiene un método `display` y no tiene ninguna superclase declarada. Esta es una situación muy simple –no están implicados ni el polimorfismo ni la herencia. A continuación, ejecutamos la instrucción:

```
v1.display();
```

Cuando se ejecuta esta instrucción, el método `display` se invoca siguiendo estos pasos:

1. Se accede a la variable `v1`.
2. Se localiza el objeto almacenado en dicha variable (siguiendo la referencia).
3. Se averigua la clase del objeto (siguiendo la referencia "instancia de").

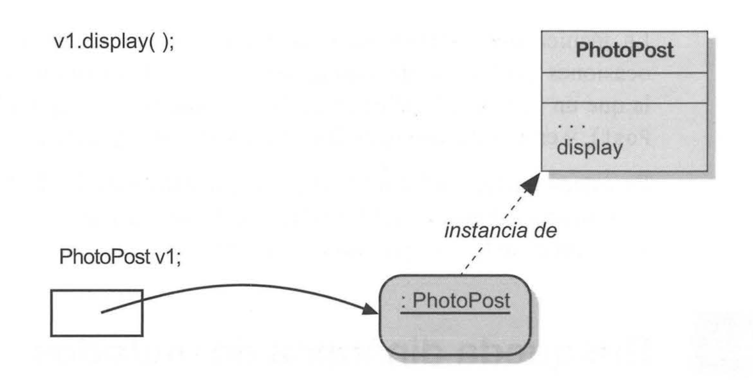


Figura 9.5
Búsqueda de métodos con un objeto simple.

4. Se localiza la implementación del método `display` en esa clase y se ejecuta.

Todo esto es bastante sencillo y nada sorprendente.

A continuación, examinemos la búsqueda del método cuando entra en juego la herencia. Este escenario es similar, pero en esta ocasión la clase `PhotoPost` tiene una superclase `Post` y el método `display` solo está definido en la superclase (Figura 9.6).

Ejecutamos la misma instrucción. La invocación del método comienza entonces de forma similar: se ejecutan de nuevo los pasos 1 a 3 del escenario anterior, pero después el proceso continúa de forma diferente:

4. No se encuentra ningún método `display` en la clase `PhotoPost`.
5. Puesto que no se ha encontrado ningún método adecuado, se busca dicho método en la superclase. Si no se encuentra ningún método en la superclase, se busca en la siguiente superclase (si existe). Esto continúa por toda la jerarquía ascendente de herencia, hasta la clase `Object`, hasta encontrar un método adecuado. Observe que en tiempo de ejecución debe siempre poder encontrarse un método que se corresponda con esa llamada, porque si no la clase no habría podido compilarse.

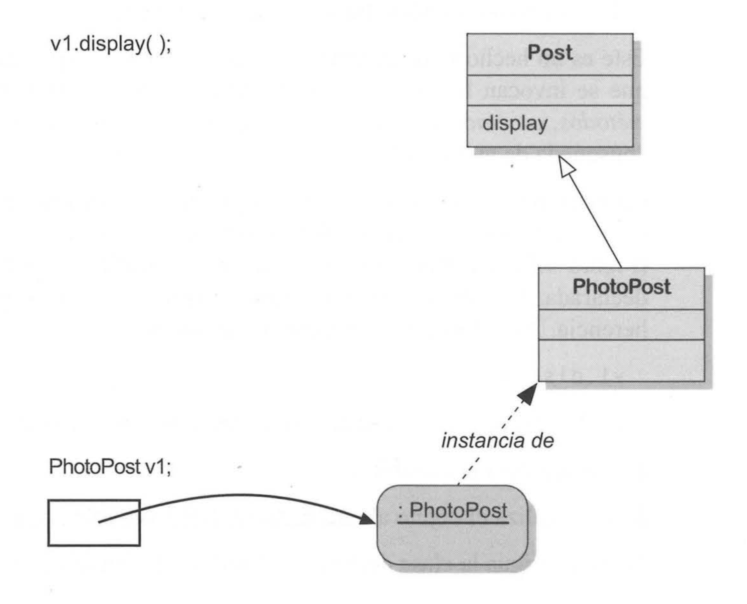


Figura 9.6
Búsqueda de método con herencia.

6. En nuestro ejemplo, se encuentra el método `display` en la clase `Post` y se ejecuta.

Este escenario ilustra cómo se produce la herencia de métodos en los objetos. Cualquier método que se encuentre en una superclase podrá ser invocado sobre un objeto de la subclase y ese método se podrá encontrar y ejecutar correctamente.

Ahora vamos al escenario más interesante: búsqueda de un método con una variable polimórfica y con el mecanismo de sustitución de métodos (Figura 9.7). El escenario es de nuevo similar al anterior, pero hay dos cambios:

- El tipo declarado de la variable `v1` ahora es `Post`, no `PhotoPost`.
- El método `display` está definido en la clase `Post` y luego redefinido (sustituido) en la clase `PhotoPost`.

Este escenario es el más importante para la comprensión del comportamiento de nuestra aplicación *network*, así como para tratar de encontrar una solución a nuestro problema del método `display`.

Los pasos con los que tiene lugar la ejecución del método son exactamente los mismos que los pasos 1 a 4 del primer escenario. Vuelva a leerlos.

Merece la pena hacer algunas observaciones:

- No se utiliza ninguna regla de búsqueda especial para la búsqueda de métodos en aquellos casos en los que el tipo dinámico no coincida con el tipo estático. El comportamiento que observamos es el resultado de la aplicación de las reglas generales.
- Quien determina qué método se encuentra en primer lugar y se ejecuta es el tipo dinámico, no el tipo estático. En otras palabras, el hecho de que el tipo declarado de `v1` sea ahora `Post` no tiene ningún efecto. La instancia con la que estamos tratando es de clase `PhotoPost` –eso es lo único que importa.
- Los métodos sustitutos de las subclases tienen precedencia sobre los métodos de la superclase. Puesto que la búsqueda de métodos comienza por la clase dinámica de la instancia (en

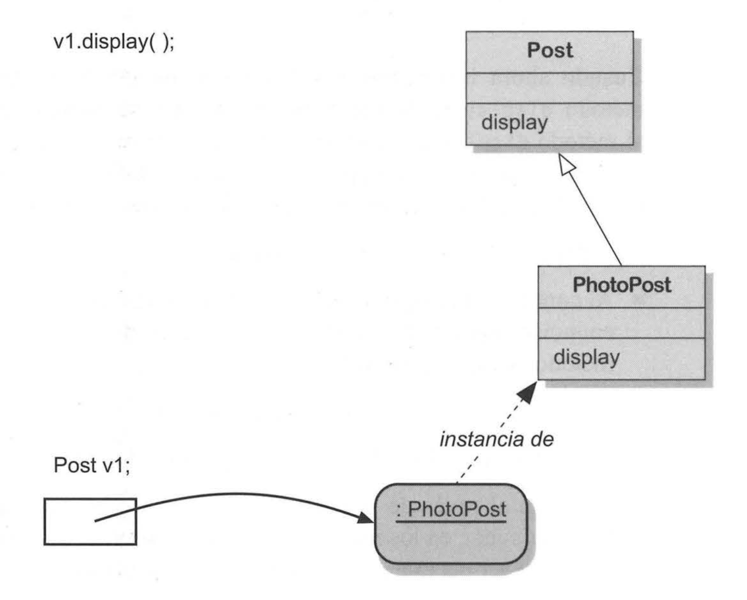


Figura 9.7
Búsqueda de método con polimorfismo y sustitución.

la parte inferior de la jerarquía de herencia), lo primero que se encuentra es la redefinición de un método, y será esa la que se ejecute.

- Cuando se sustituye un método, solo se ejecuta la última versión (la que esté más abajo en la jerarquía de herencia). Las versiones del mismo método contenidas en cualquiera de las superclases no se ejecutan automáticamente.

Esto explica el comportamiento que hemos observado en nuestro proyecto *network*. Solo se ejecutan los métodos `display` de las subclases (`MessagePost` y `PhotoPost`) a la hora de imprimir las publicaciones, lo que hace que dispongamos de listados incompletos. En la siguiente sección veremos cómo corregir esto.

9.5 Llamada a `super` en los métodos

Ahora que conocemos en detalle cómo se ejecutan los métodos sustitutos, podemos entender cuál es la solución al problema. Es fácil ver que lo que queremos es que cada llamada al método `display` de, por ejemplo, un objeto `PhotoPost`, haga que se ejecuten tanto el método `display` de la clase `Post` como el de la clase `PhotoPost` para ese mismo objeto. Entonces se imprimirán correctamente todos los detalles. (Posteriormente en el capítulo presentaremos una solución diferente.)

Esto es, de hecho, muy fácil de conseguir. Podemos utilizar la estructura `super`, con la que ya nos hemos encontrado en el contexto de los constructores en el Capítulo 8. El Código 9.2 ilustra esta idea con el método `display` de la clase `PhotoPost`.

Código 9.2 Redefinición del método con una llamada a `super`.

```
public void display()
{
    super.display();
    System.out.println("    [" + filename + "]");
    System.out.println("    " + caption);
}
```

Cuando ahora invocamos `display` sobre un objeto `PhotoPost`, inicialmente se llamará al método `display` de la clase `PhotoPost`. En su primera instrucción, este método invocará al método `display` de la superclase, que imprimirá la información general de la publicación. Cuando el método de la superclase devuelva el control, las instrucciones restantes del método de la subclase imprimirán los campos distintivos de la clase `PhotoPost`.

Hay tres detalles que merece la pena resaltar:

- Al contrario de lo que sucede en el caso de las llamadas `super` en los constructores, hay que enunciar explícitamente el nombre del método de la superclase. Una llamada `super` en un método siempre tiene la forma

  `super.`*nombre-método* (*parámetros*)

  Por supuesto, la lista de parámetros puede estar vacía.

- También al revés de la regla que se aplica a las llamadas `super` en los constructores, la llamada `super` en los métodos puede estar situada en cualquier lugar dentro del método. No tiene que ser necesariamente la primera instrucción.

- Y también al revés que en el caso de las llamadas `super` en los constructores, no se generará ninguna llamada `super` automática y tampoco es obligatorio emplear una llamada `super`; es una llamada enteramente opcional. Por tanto, el comportamiento predeterminado consiste en que el método de la subclase oculta completamente (es decir, sustituye) la versión de ese mismo método contenida en la superclase.

Ejercicio 9.3 Modifique su última versión del proyecto *network* para incluir la llamada `super` en el método `display`. Pruebe su solución. ¿Se comporta de la forma esperada? ¿Puede haber algún problema con esta solución?

Merece la pena reiterar lo que hemos ilustrado en el Ejercicio 8.6: que en ausencia del mecanismo de sustitución de métodos, los miembros no privados de una superclase son directamente accesibles desde sus subclases sin necesidad de ninguna sintaxis especial. Solo es necesario hacer una llamada `super` cuando haga falta acceder a la versión existente en la superclase de un método *sustituido*.

Si ha completado el Ejercicio 9.3, habrá observado que esta solución funciona, pero que todavía no es perfecta. Imprime todos los detalles pero en un orden distinto del que deseábamos. Corregiremos este último problema más adelante en el capítulo.

9.6 Polimorfismo de métodos

Concepto:

Polimorfismo de métodos Las llamadas a métodos en Java son polimórficas. La misma llamada a método puede invocar diferentes métodos en distintos momentos, dependiendo del tipo dinámico de la variable usada para hacer dicha llamada.

Lo que acabamos de explicar en las secciones anteriores (Secciones 9.2–9.5) es otra forma de polimorfismo. Es lo que se conoce con el nombre de *despacho de métodos polimórficos* (o para abreviar *polimorfismo de métodos*).

Recuerde que una variable polimórfica es aquella que puede almacenar objetos de diversos tipos (toda variable de objeto en Java es potencialmente polimórfica). De manera similar, las llamadas a métodos Java son polimórficas, porque pueden invocar diferentes métodos en distintos momentos. Por ejemplo, la instrucción

```
post.display();
```

podría invocar el método `display` de `MessagePost` en un determinado momento y el método `display` de `PhotoPost` en otro, dependiendo del tipo dinámico de la variable `post`.

9.7 Métodos de `Object`: `toString`

En el Capítulo 8, hemos mencionado que la superclase universal, `Object`, implementa algunos métodos que forman parte, como consecuencia, de todos los objetos. El más interesante de estos métodos es `toString`, que vamos a ver aquí (si quiere conocer más detalles, puede buscar la interfaz de `Object` en la documentación de la librería estándar).

Ejercicio 9.4 Busque `toString` en la documentación de la librería. ¿Cuáles son sus parámetros? ¿Cuál es su tipo de retorno?

Concepto:

Todo objeto en Java tiene un método `toString` que puede utilizarse para devolver una representación de ese objeto en forma de `string`. Normalmente, para que este método sea útil, los objetos deben sustituirlo por una implementación propia.

El propósito del método `toString` es crear una representación de un objeto en forma de cadena de caracteres. Esto resulta útil para cualquier objeto que vaya a ser representado en forma textual en la interfaz de usuario, pero también es útil para todos los demás objetos, que con ese método podrán ser impresos de manera fácil, por ejemplo con propósitos de depuración.

La implementación predeterminada de `toString` en la clase `Object` no puede suministrar una gran cantidad de detalle. Por ejemplo, si invocamos `toString` sobre un objeto `PhotoPost`, recibiremos una cadena de caracteres similar a esta:

```
PhotoPost@6acdd1
```

El valor de retorno simplemente muestra el nombre de la clase del objeto y un número mágico.[2]

Ejercicio 9.5 Puede probar este método fácilmente. Cree un objeto de clase `PhotoPost` en su proyecto y luego invoque el método `toString` del submenú de `Object`, en el menú emergente del objeto.

Para que este método sea más útil, lo que normalmente haremos será sustituirlo en nuestras propias clases. Por ejemplo, podemos definir el método `display` de `Post` en términos de una llamada a su método `toString`. En este caso, el método `toString` no imprimiría los detalles, se limitaría a crear una cadena de caracteres con el texto. El Código 9.3 muestra el código fuente modificado.

Código 9.3
Método `toString` para `Post` y `MessagePost`.

```
public class Post
{
    ...
    public String toString()
    {
        String text = username + "\n" + timeString(timestamp);
        if(likes > 0) {
            text += "  " + likes + " people like this.\n";
        }
        else {
            text += "\n";
        }

        if(comments.isEmpty()) {
            return text + "   No comments.\n";
        }
        else {
            return text + "   " + comments.size() +
                   " comment(s). Click here to view.\n";
```

[2] El número mágico es, de hecho, la dirección de memoria en la que está almacenado el objeto. No resulta demasiado útil, salvo para establecer la identidad del objeto. Si este número es el mismo en dos llamadas, querrá decir que estamos ante un mismo objeto. Si es diferente, tendremos dos objetos distintos.

Código 9.3
(continuación)
Método `toString` para `Post` y `MessagePost`.

```
        }
    }

    public void display()
    {
        System.out.println(toString());
    }
}

public class MessagePost extends Post
{
    ...
    public String toString()
    {
        return super.toString() + message + "\n";
    }

    public void display()
    {
        System.out.println(toString());
    }
}
```

En último término, lo que nos gustaría es eliminar completamente los métodos `display` de estas clases. Una gran ventaja de definir solamente un método `toString` es que no decimos en las clases `Post` qué es exactamente lo que hay que hacer con el texto de la descripción. La versión original siempre imprimía el texto en el terminal de salida. Ahora, cualquier cliente (por ejemplo, la clase `NewsFeed`) será libre de hacer lo que quiera con ese texto. Bien mostrarlo en un área texto de una interfaz gráfica de usuario; guardarlo en un archivo; enviarlo a través de una red; mostrarlo en un explorador web o, como antes, o imprimirlo en el terminal.

La instrucción utilizada en el cliente para imprimir la publicación tiene ahora el aspecto siguiente:

```
System.out.println(post.toString());
```

De hecho, los métodos `System.out.print` y `System.out.println` son especiales a este respecto: si el parámetro de uno de los métodos no es un objeto `String`, entonces el método invoca automáticamente al método `toString` del objeto. Por tanto, no necesitamos escribir esa llamada explícitamente, y podríamos en su lugar limitarnos a escribir

```
System.out.println(post);
```

Considere ahora la versión modificada del método `show` de la clase `NewsFeed` mostrada en el Código 9.4. En esta versión, hemos eliminado la llamada a `toString`, ¿cree que se compilará y ejecutará correctamente?

Código 9.4
Nueva versión del método `show` de `NewsFeed`.

```
public class NewsFeed
{
    // Se omiten campos, constructores y otros métodos.

    /**
     * Mostrar la fuente de noticias. Actualmente: imprimir los
     * detalles en el terminal. (Para más adelante: sustituir esto
     * por una visualización en un explorador web.)
     */
    public void show()
    {
        for(Post post : posts) {
            System.out.println(post);
        }
    }
}
```

De hecho, el método *sí que funciona* de la forma esperada. Si puede explicar este ejemplo detalladamente, probablemente quiera decir que ¡ya comprende la mayor parte de los conceptos que hemos presentado en este capítulo y en el anterior! He aquí una explicación detallada de la única instrucción `println` que podemos ver dentro del bucle.

- El bucle for-each itera a través de todas las publicaciones y las almacena en una variable que tiene el tipo estático `Post`. El tipo dinámico puede ser `MessagePost` o `PhotoPost`.
- Puesto que este objeto está siendo impreso en `System.out` y no es de tipo `String`, se invoca automáticamente su método `toString`.
- La invocación de este método es válida únicamente porque la clase `Post` (¡el tipo estático!) tiene un método `toString`. (Recuerde: la comprobación de tipos se lleva a cabo con el tipo estático. Esta llamada no estaría permitida si la clase `Post` no tuviera un método `toString`. Sin embargo, el método `toString` de la clase `Object` garantiza que este método esté siempre disponible para cualquier clase.)
- La salida se muestra apropiadamente con todos los detalles, ya que cada posible tipo dinámico (`MessagePost` y `PhotoPost`) sustituye el método `toString` y el mecanismo de búsqueda de métodos garantiza que siempre se ejecute el método redefinido.

El método `toString` suele ser útil para propósitos de depuración. A menudo, resulta bastante cómodo que los objetos puedan imprimirse fácilmente con un formato razonable. La mayoría de las clases de librería Java sustituyen `toString` (por ejemplo, todas las colecciones se pueden imprimir de esta manera), y a menudo es una buena idea sustituir también este método en nuestras propias clases.

9.8 Igualdad entre objetos: `equals` y `hashCode`

A menudo es necesario determinar cuándo dos objetos son "el mismo" objeto. La clase `Object` define dos métodos, `equals` y `hashCode`, que están estrechamente relacionados con la tarea de determinar la similitud. En realidad, tenemos que tener cuidado al utilizar frases tales como "el

mismo objeto", porque esa frase puede significar dos cosas completamente distintas cuando hablamos de objetos. En ocasiones, lo que deseamos es saber si dos variables diferentes están haciendo referencia al mismo objeto. Esto es exactamente lo que sucede cuando se pasa una variable de objeto como parámetro de un método: solo hay un único objeto, pero tanto la variable original como la variable del parámetro hacen referencia a él. Lo mismo sucede cuando se asigna una variable de objeto a otra. Estas situaciones dan lugar a lo que se conoce con el nombre de *igualdad de referencias*. La igualdad de referencia se comprueba utilizando el operador ==. Por tanto, la siguiente comprobación devolverá `true` si `var1` y `var2` están haciendo referencia al mismo objeto (o ambas son `null`), y devolverá `false` si están haciendo referencia a alguna otra cosa:

```
var1 == var2
```

La igualdad de referencia no tiene en cuenta en absoluto el *contenido* de los objetos a los que se hace referencia, sino que se limita a comprobar si hay un único objeto al que están haciendo referencia dos variables distintas o dos objetos distintos. Esa es la razón por la que también nos vemos obligados a definir la *igualdad de contenidos*, que es distinta de la igualdad de referencias. Lo que una comprobación de la igualdad de contenidos pregunta es si dos objetos son iguales internamente, es decir, si los estados internos de los dos objetos coinciden. Esa es la razón por la que rechazamos en el Capítulo 5 utilizar la igualdad de referencias a la hora de realizar comparaciones de cadenas de caracteres.

Lo que la igualdad de contenidos entre dos objetos concretos significa es algo que está definido por la clase de los objetos. Es ahí donde hacemos uso del método `equals` que toda clase hereda de la superclase `Object`. Si necesitamos definir qué quiere decir que dos objetos sean iguales de acuerdo con sus estados internos, entonces tendremos que sustituir el método `equals`, lo que nos permitirá escribir comprobaciones como

```
var1.equals(var2)
```

Esto es así porque lo que hace el método `equals` heredado de la clase `Object` es comprobar que existe una igualdad de referencias. Su aspecto es similar al siguiente:

```
public boolean equals(Object obj)
{
    return this == obj;
}
```

Puesto que la clase `Object` no tiene ningún campo, no hay ningún estado que comparar y este método no puede, obviamente, prever de antemano los campos que vayan a estar presentes en las subclases.

La forma de comprobar la igualdad de contenidos entre dos objetos consiste en verificar si los valores de sus dos conjuntos de campos son iguales. Observe, sin embargo, que el parámetro del método `equals` es de tipo `Object`, por lo que esa comprobación de los campos solo tendrá sentido si estamos comparando campos del mismo tipo. Esto significa que primero tenemos que verificar que el tipo del objeto pasado como parámetro sea igual que el del objeto con el que se está comparando. He aquí cómo se podría escribir ese método en la clase `Student` del proyecto *lab-classes* del Capítulo 1:

```
public boolean equals(Object obj)
{
    if(this == obj) {
        return true;    // Igualdad de referencias.
    }
    if(!(obj instanceof Student)) {
        return false;   // No son del mismo tipo.
    }
    // Acceder a los campos del otro estudiante.
    Student other = (Student) obj;
    return name.equals(other.name) &&
           id.equals(other.id) &&
           credits == other.credits;
}
```

La primera de las comprobaciones pretende simplemente mejorar la eficiencia; si lo que se le ha pasado al objeto para hacer una comparación es una referencia a sí mismo, entonces sabemos que existirá una igualdad de contenidos. La segunda comprobación verifica que estamos comparando dos estudiantes. Si no es así, entonces decidimos que los dos objetos no pueden ser iguales. Habiendo establecido que lo que tenemos es otro estudiante, utilizamos un *cast* y otra variable del tipo adecuado para poder acceder apropiadamente a sus detalles. Finalmente, aprovechamos el hecho de que los elementos privados de un objeto son directamente accesibles por parte de una instancia de la misma clase; esto es esencial en situaciones como esta, porque no necesariamente va a haber definidos métodos selectores para todos los campos privados de una clase. Observe que hemos usado, como es lógico, comprobaciones de igualdad de contenidos en lugar de comprobaciones de igualdad de las referencias al comparar los campos de objeto denominados `name` e `id`.

No siempre será necesario comparar todos los campos de dos objetos para concluir que son iguales. Por ejemplo, si estamos seguros de que a cada objeto `Student` se le asigna un valor `id` distintivo, entonces no necesitamos comprobar además los campos `name` y `credits`. En ese caso, sería posible reducir la instrucción final del segmento de código anterior a

```
return id.equals(other.id);
```

Siempre que se sustituya el método `equals`, es necesario sustituir también el método `hashCode`. Las estructuras de datos como `HashMap` y `HashSet` utilizan el método `hashCode` para mejorar la eficiencia en el almacenamiento y la búsqueda de objetos en esos tipos de colecciones. Básicamente, el método `hashCode` devuelve un valor entero que representa un objeto. Gracias a la implementación predeterminada existente en `Object`, los diferentes objetos tendrán valores `hashCode` distintos.

Existe una importante vinculación entre los métodos `equals` y `hashCode` en el sentido de que dos objetos que sean iguales, según determine una llamada a `equals`, deben devolver valores idénticos de `hashCode`. Esta funcionalidad, que se debe cumplir obligatoriamente, se especifica en la descripción de `hashCode`, en la documentación de la API de la clase `Object`.[3] Queda fuera del alcance de este libro describir en detalle una técnica adecuada para el cálculo de los códigos hash, pero al lector interesado le recomendamos el libro *Effective Java* de

[3] Observe que no es obligatorio que dos objetos distintos devuelvan siempre códigos hash diferentes.

Joshua Bloch, cuya técnica es la que usamos aquí.[4] Esencialmente, hay que calcular un valor entero haciendo uso de los valores de los campos que se comparan mediante el método `equals` sustituido. He aquí un método `hashCode` hipotético que utiliza los valores de un campo entero denominado `count` y de un campo de tipo `String` denominado `name` para calcular el código:

```
public int hashCode()
{
    int result = 17;    // Un valor inicial arbitrario.
    // Hacer que el valor calculado dependa del orden en
    // que se procesen los campos.
    result = 37 * result + count;
    result = 37 * result + name.hashCode();
    return result;
}
```

9.9 Acceso protegido

Concepto:

Declarar un campo o un método **protegido** permite acceder directamente a él desde las subclases (directas o indirectas).

En el Capítulo 8, hemos indicado que las reglas sobre la visibilidad pública y privada de los miembros de una clase se aplican entre una subclase y su superclase, al igual que entre clases situadas en diferentes jerarquías de herencia. Esto puede ser algo restrictivo, porque la relación entre una superclase y sus subclases es, obviamente, más estrecha que con otras clases. Por esta razón, los lenguajes orientados a objetos suelen definir un nivel de acceso que está situado a medio camino entre la completa restricción propia del acceso privado y la completa disponibilidad propia del acceso público. En Java, este tipo de acceso se denomina *acceso protegido* y se proporciona mediante la palabra clave `protected`, que es una alternativa a `public` y a `private`. El Código 9.5 muestra un ejemplo de un método selector protegido que podríamos añadir a la clase `Post`.

Código 9.5
Un ejemplo de un método protegido.

```
protected long getTimeStamp()
{
    return timestamp;
}
```

El acceso protegido permite acceder a los campos o métodos situados dentro de la misma clase y también desde todas sus subclases, pero no desde las clases restantes.[5] El método `getTimeStamp` mostrado en el Código 9.5 podría ser invocado desde la clase `Post` o desde cualquiera de sus subclases, pero no desde otras clases. La Figura 9.8 ilustra este hecho. Los óvalos mostrados en el diagrama indican los grupos de clases que pueden acceder a los distintos miembros de la clase `AlgunaClase`.

---

[4] En el momento de escribir estas líneas podían encontrarse capítulos de ejemplo que contienen material relevante para este tema en la dirección `http://java.sun.com/developer/Books/effectivejava/`.

[5] En Java, esta regla no es tan clara como aquí se describe, porque Java incluye un nivel adicional de visibilidad denominado *nivel de paquete*, pero que no tiene asociado una palabra clave. No vamos a tratar este tema aquí y además es más general considerar el acceso protegido como algo concebido para la relación especial existente entre una superclase y una subclase.

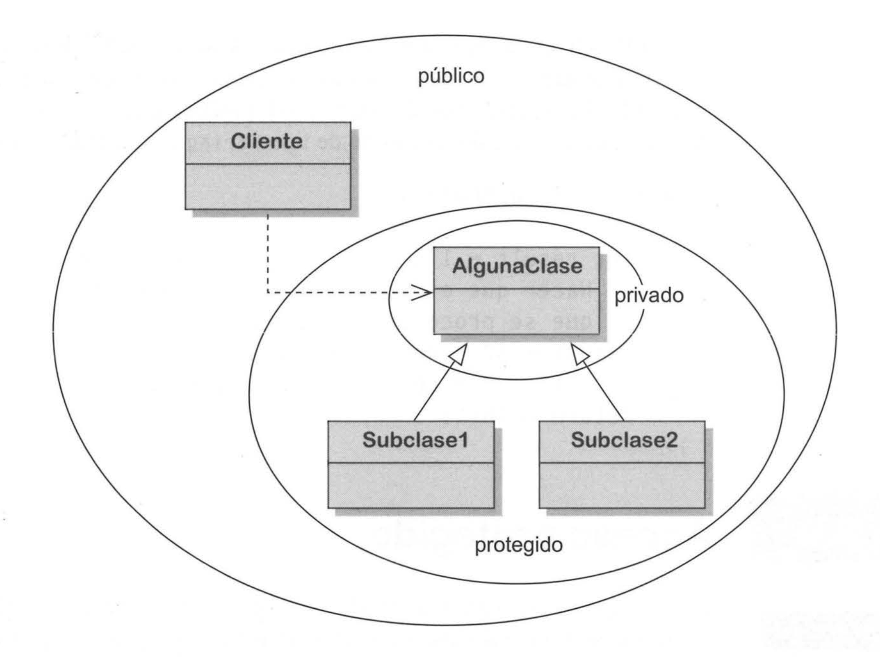


Figura 9.8
Niveles de acceso: privado, protegido y público.

Aunque el acceso protegido puede aplicarse a cualquier miembro de una clase, suele reservarse para métodos y constructores. No se suele aplicar a los campos, porque eso debilitaría la encapsulación. Siempre que sea posible, los campos mutables de las superclases deberían seguir siendo privados. No obstante, existen casos válidos ocasionales en los que es deseable que las subclases dispongan de acceso directo. La herencia representa una forma mucho más estrecha de acoplamiento que la de las relaciones normales con las clases clientes.

La herencia asocia las clases más estrechamente, por lo que cualquier modificación de la superclase tiene más posibilidades de hacer que la subclase deje de funcionar. Esto debe tenerse en cuenta a la hora de diseñar las clases y sus relaciones.

Ejercicio 9.6 La versión de `display` mostrada en el Código 9.2 genera la salida mostrada en la Figura 9.9. Reordene las instrucciones del método en su versión del proyecto *network* para que imprima los detalles tal como se ilustra en la Figura 9.10.

Figura 9.9
Posible salida de `display`: llamada a la superclase al principio de `display` (las áreas sombreadas son imprimidas por el método de la superclase).

```
Leonardo da Vinci
40 seconds ago - 2 people like this.
  No comments.
Had a great idea this morning.
But now I forgot what it was. Something to do with flying...
```

Figura 9.10
Salida alternativa de `display` (las áreas sombreadas son imprimidas por el método de la superclase).

```
Had a great idea this morning.
But now I forgot what it was. Something to do with flying...
Leonardo da Vinci
40 seconds ago - 2 people like this.
  No comments.
```

Ejercicio 9.7 Tener que usar una llamada a la superclase en `display` es algo restrictivo en lo que respecta a las maneras con las que podemos dar formato a la salida, porque depende de la forma en que la superclase formatee sus campos. Haga los cambios necesarios en la clase `Post` y en le método `display` de `MessagePost` para que genere la salida mostrada en la Figura 9.11. Cualquier cambio que realice en la clase `Post` debe ser solo visible para sus subclases. *Sugerencia:* puede añadir métodos selectores protegidos para hacer esto.

Figura 9.11
Salida de `display` en la que se mezclan detalles generados por la subclase y la superclase (las áreas sombreadas representan los detalles de la superclase.)

```
Leonardo da Vinci
Had a great idea this morning.
But now I forgot what it was. Something to do with flying...
40 seconds ago - 2 people like this.
  No comments.
```

9.10 El operador `instanceof`

Una de las consecuencia de la introducción de la herencia en el proyecto *network* ha sido que la clase `NewsFeed` solo sabe de la existencia de objetos `Post` y es incapaz de distinguir entre publicaciones de mensajes y publicaciones fotográficas. Esto nos ha permitido almacenar todos los tipos de publicaciones en una lista.

Sin embargo, suponga que quisiéramos extraer solamente de las lista las publicaciones de mensajes o las publicaciones fotográficas; ¿cómo podríamos hacer esto? o ¿qué pasa si lo que queremos es buscar un mensaje de un determinado autor? Esto no sería un problema si la clase `Post` define un método a `getAuthor`, pero eso haría que localizáramos publicaciones tanto de mensajes como de fotografías. ¿Qué pasa si el tipo de publicación devuelta es relevante?

Hay ocasiones en las que necesitamos averiguar el tipo dinámico distintivo de un objeto, en lugar de limitarnos a tratar con un supertipo compartido. Para estos casos, Java proporciona el operador `instanceof`. Este operador comprueba si un objeto determinado es, directa o indirectamente, una instancia de una determinada clase. La comprobación

```
obj instanceof MyClass
```

devuelve `true` si el tipo dinámico de `obj` es `MyClass` o cualquier subclase de `MyClass`. El operando izquierdo es siempre una referencia a objeto y el operando derecho es siempre el nombre de una clase. Por tanto,

```
post instanceof MessagePost
```

devuelve `true`, por ejemplo, solo si `post` es una instancia de `MessagePost`, y no es una instancia de `PhotoPost`.

La utilización del operador `instanceof` suele ir seguida inmediatamente por un cast de la referencia a objeto, para transformarla en el tipo identificado. Por ejemplo, he aquí un fragmento de código que permite identificar todas las publicaciones de mensajes en una lista de publicaciones y almacenarlas en una lista separada.

```
ArrayList<MessagePost> messages = new ArrayList<MessagePost>();
for(Post post : posts) {
    if(post instanceof MessagePost) {
        messages.add((MessagePost) post);
    }
}
```

Debería quedar claro que el cast que se utiliza aquí no altera el objeto publicación en ningún sentido, porque acabamos de determinar que ya es un objeto `MessagePost`.

9.11 Otro ejemplo de herencia con sustitución

Para analizar otro ejemplo de un uso similar de la herencia, volvamos a un proyecto del Capítulo 6: el proyecto *zuul*. En el juego *world-of-zuul*, utilizábamos un conjunto de objetos `Room` para crear una escena para un juego simple. Uno de los ejercicios de la parte final del capítulo sugería implementar una sala transportadora (una sala que nos proyecte a una ubicación aleatoria del juego, si tratamos de entrar o salir de ella). Vamos a volver aquí sobre ese ejercicio, porque su solución puede aprovecharse enormemente del mecanismo de herencia. Si no recuerda bien ese proyecto, vuelva a echar un vistazo rápido al Capítulo 6 o examine su propio proyecto *zuul*.

No hay una solución única para esta tarea. Se pueden concebir muchas soluciones distintas y se puede conseguir que funcionen todas ellas. Sin embargo, hay algunas soluciones mejores que otras: puede que sean más elegantes, más fáciles de leer y más fáciles de mantener y de ampliar.

Suponga que queremos implementar esta tarea de modo que el jugador sea transportado automáticamente a una sala aleatoria cuando trate de salir de la sala transportadora mágica. La solución más sencilla que primero se le ocurre a muchas personas consiste en resolver este problema dentro de la clase `Game`, que implementa los comandos del jugador. Uno de los comandos es *go*, que está implementado en el método `goRoom`. En este método, utilizábamos la siguiente instrucción como sección central del código:

```
nextRoom = currentRoom.getExit(direction);
```

Esta instrucción extrae de la sala actual la sala adyacente en la dirección en la que nos queramos mover. Para añadir nuestro transporte mágico, podríamos modificar ese código de una forma similar a esta:

```
if(currentRoom.getName().equals("Transporter room")) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

La idea subyacente a esta modificación es bastante simple: nos limitamos a comprobar si nos encontramos en la sala transportadora. Si es así, entonces determinamos la siguiente sala seleccionando una sala aleatoria (por supuesto, tenemos que implementar de alguna manera el método `getRandomRoom`); en caso contrario, hacemos lo mismo que antes.

Aunque esta solución funciona, tiene varias desventajas. La primera es que no es conveniente utilizar cadenas de texto como nombre de la sala, para identificar la sala. Imagine que alguien quisiera traducir nuestro juego a otro idioma, por ejemplo, al alemán. En ese caso, podrían cambiar los nombres de las salas y *Transporter room* se transformaría en *Transporterraum*, con lo que el juego dejaría de repente de funcionar. Este es un caso claro de problema de mantenibilidad.

La segunda solución, que es ligeramente mejor, consistiría en utilizar una variable de instancia en lugar del nombre de la sala para identificar la sala transportadora. La solución tendría el siguiente aspecto:

```
if(currentRoom == transporterRoom) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

Esta vez, suponemos que disponemos de una variable de instancia `transporterRoom` de la clase `Room`, en la que almacenamos la referencia a nuestra sala transportadora.[6] Ahora, la comprobación es independiente del nombre de la sala. Esto está un poco mejor.

Sin embargo, sigue existiendo una posible mejora. Podemos entender las limitaciones de esta solución si pensamos en otra posible modificación de mantenimiento. Imagine que queremos añadir dos salas transportadoras más, de modo que nuestro juego disponga de tres ubicaciones transportadoras diferentes.

Un aspecto muy elegante de nuestro diseño existente era que podíamos definir el mapa en un único punto, siendo el resto del juego completamente independiente del mismo. Podíamos cambiar fácilmente la disposición de las salas y todo seguía funcionando –¡Esto es una gran ventaja de cara a la mantenibilidad! Sin embargo, en nuestra solución actual esto deja ya de ser cierto. Si añadimos dos nuevas salas transportadoras, necesitamos añadir dos variables de instancia adicionales o una matriz (para almacenar referencias a esas salas), y tenemos que modificar nuestro método `goRoom` para añadir una comprobación relativa a esas salas. En términos de facilidad de modificación, hemos retrocedido con respecto a la solución anterior.

Por tanto, la cuestión es la siguiente: ¿podemos encontrar una solución que no requiera modificar la implementación de los comandos cada vez que añadamos una nueva sala transportadora? Veamos una posible idea.

Podemos añadir un método `isTransporterRoom` en la clase `Room`. De esta forma, el objeto `Game` no necesita recordar todas las salas transportadoras –son las propias salas las que se encargan de ello. Cuando se crean las salas, puede definirse un indicador booleano que especifique si una determinada sala es una sala transportadora. El método `goRoom` podría entonces utilizar el siguiente segmento de código:

---

[6] Asegúrese de entender por qué lo más apropiado en este caso es comprobar la igualdad de referencias.

```
if(currentRoom.isTransporterRoom()) {
    nextRoom = getRandomRoom();
}
else {
    nextRoom = currentRoom.getExit(direction);
}
```

Ahora podemos añadir tantas salas transportadoras como queramos; no hay necesidad de efectuar ningún cambio adicional en la clase `Game`. Sin embargo, la clase `Room` tiene un campo adicional cuyo valor solo es necesario debido a la naturaleza de una o dos de las instancias. El código para casos especiales como este es un indicador típico de que el diseño de clases tiene una debilidad. Asimismo, esta solución no es muy ampliable, en el caso de que decidamos introducir más tipos de salas especiales, requiriendo cada uno de ellos su propio campo indicador y su propio método selector.[7]

Con la herencia, podemos mejorar la solución e implementar otra que sea aun más flexible que esta. Podemos implementar una clase `TransporterRoom` como subclase de la clase `Room`. En esta nueva clase, sustituiremos el método `getExit` y modificaremos su implementación para que devuelva una sala aleatoria.

```
public class TransporterRoom extends Room
{
    /**
     * Devuelve una sala aleatoria, independiente
     * del parámetro direction.
     * @param direction Ignorado.
     * @return Una sala aleatoria.
     */
    public Room getExit(String direction)
    {
        return findRandomRoom();
    }

    /*
     * Selecciona una sala aleatoria.
     * @return Una sala aleatoria.
     */
    private Room findRandomRoom()
    {
        ... // implementación omitida
    }
}
```

La elegancia de esta solución radica en el hecho de que no hace falta modificar en absoluto las clases `Game` o `Room` originales. Podemos simplemente añadir esta clase al juego existente y el método `goRoom` continuará funcionando tal cual está. Añadir la creación de una instancia

[7] También podríamos pensar en utilizar `instanceof`, pero lo que queremos resaltar aquí es que ninguna de estas ideas es la mejor.

de TransporterRoom al proceso de creación del mapa es (casi) suficiente para hacer que esta solución funcione. Observe también que la nueva clase no necesita ningún indicador para señalar su naturaleza especial –su propio tipo y su comportamiento distintivo suministran esa información.

Dado que TransporterRoom es una subclase de Room, puede utilizarse en cualquier lugar donde se espere un objeto Room. Por tanto, puede emplearse como sala adyacente de otra sala o puede almacenarse en el objeto Game como sala actual.

Lo que hemos omitido, por supuesto, es la implementación del método findRandomRoom. En realidad, esto probablemente se haría mejor en una clase separada (por ejemplo, RoomRandomizer) que en la propia clase TransporterRoom. Dejamos esto planteado como ejercicio para el lector.

Ejercicio 9.8 Implemente una sala transportadora utilizando el mecanismo de herencia en su versión del proyecto *zuul*.

Ejercicio 9.9 Explique cómo podría utilizarse la herencia en el proyecto *zuul* para implementar una clase jugador y una clase monstruo.

Ejercicio 9.10 ¿Podría (o debería) utilizarse la herencia para crear una relación de herencia (superclase, subclase o clase hermana) entre un personaje del juego y un elemento?

9.12 Resumen

Al tratar con clases y subclases y con variables polimórficas, tenemos que distinguir el tipo estático y el tipo dinámico de una variable. El tipo estático es el tipo declarado, mientras que el tipo dinámico es el tipo del objeto actualmente almacenado en la variable.

La comprobación de tipos la lleva a cabo el compilador empleando el tipo estático, mientras que el mecanismo de búsqueda de métodos en tiempo de ejecución utiliza el tipo dinámico. Esto permite crear estructuras muy flexibles sustituyendo los métodos. Aun cuando estemos utilizando una variable del supertipo para hacer una llamada a un método, el mecanismo de sustitución nos permite garantizar que se invoquen métodos especializados para cada subtipo concreto. Esto garantiza que los objetos de las diferentes clases puedan reaccionar de manera distinta a una misma llamada a método.

Al implementar métodos sustitutos, puede utilizarse la palabra clave super para invocar la versión del método en la superclase. Si se declaran los campos o los métodos con el modificador de acceso protected, las subclases estarán autorizadas a acceder a los mismos, pero otras clases no lo estarán.

Términos introducidos en el capítulo

tipo estático, tipo dinámico, sustitución de métodos, redefinición, búsqueda de métodos, despacho de métodos, polimorfismo de métodos, acceso protegido

Resumen de conceptos

- **tipo estático** El tipo estático de una variable *v* es el tipo tal como está declarado en el código fuente, en la instrucción de declaración de la variable.
- **tipo dinámico** El tipo dinámico de una variable *v* es el tipo del objeto que está almacenado actualmente en *v*.
- **sustitución de métodos** Una subclase puede sustituir la implementación de un método. Para ello, la subclase declara un método con la misma signatura que la superclase, pero con un cuerpo del método diferente. El método sustituto tiene precedencia en las llamadas a método efectuadas sobre los objetos de la subclase.
- **polimorfismo de métodos** Las llamadas a métodos en Java son polimórficas. La misma llamada a método puede invocar diferentes métodos en distintos momentos, dependiendo del tipo dinámico de la variable usada para hacer dicha llamada.
- **toString** Todo objeto en Java tiene un método `toString` que puede utilizarse para devolver una representación de ese objeto en forma de `string`. Normalmente, para que este método sea útil, los objetos deben sustituirlo por una implementación propia.
- **acceso protegido** Declarar un campo o un método protegido permite acceder directamente a él desde las subclases (directas o indirectas).

Ejercicio 9.11 Suponga que nos encontramos las siguientes líneas de código:

```
Device dev = new Printer();
dev.getName();
```

`Printer` es una subclase de `Device`, que representa un dispositivo de computadora. ¿Cuál de estas clases debe disponer de una definición del método `getName` para que este código se compile?

Ejercicio 9.12 En la misma situación que en el ejercicio anterior, si ambas clases tienen una implementación de `getName`, ¿cuál se ejecutará?

Ejercicio 9.13 Suponga que escribimos una clase `Student` que no dispone de ninguna superclase declarada. Suponga también que no escribimos un método `toString`. Considere las siguientes líneas de código:

```
Student st = new Student();
String s = st.toString();
```

¿Podrán compilarse estas líneas? En caso afirmativo, ¿qué sucederá exactamente cuando trate de ejecutarlas?

Ejercicio 9.14 En la misma situación que antes (clase `Student` sin método `toString`), ¿podrán compilarse estas líneas? ¿Por qué?

```
Student st = new Student();
System.out.println(st);
```

Ejercicio 9.15 Suponga que su clase `Student` sustituye el método `toString`, para que este devuelva el nombre del estudiante. Ahora suponga que disponemos de una lista de estudiantes. ¿Podrá compilarse el

siguiente código? En caso negativo, ¿por qué no? En caso afirmativo, ¿qué se imprimiría? Explique en detalle lo que sucede.

```
for(Object st : myList) {
    System.out.println(st);
}
```

Ejercicio 9.16 Escriba unas cuantas líneas de código que provoquen una situación en la que una variable `x` tenga el tipo estático `T` y el tipo dinámico `D`.

CAPÍTULO

10 Técnicas de abstracción adicionales

Principales conceptos explicados en el capítulo:

- clases abstractas
- interfaces
- herencia múltiple

Estructuras Java explicadas en este capítulo:

`abstract, implements, interface`

En este capítulo vamos a examinar técnicas adicionales relacionadas con la herencia que pueden utilizarse para mejorar las estructuras de clases y para facilitar la mantenibilidad y la ampliabilidad. Estas técnicas incorporan un método mejorado de representación de las abstracciones en los programas orientados a objetos.

En los dos capítulos anteriores hemos explicado los aspectos más importantes de la herencia en relación con el diseño de aplicaciones, pero hasta el momento hemos ignorado otros usos y problemas más avanzados. Ahora completaremos la imagen global con un ejemplo más sofisticado.

El proyecto que vamos a utilizar en este capítulo es una simulación. La usaremos para estudiar de nuevo la herencia y ver cómo dicho mecanismo nos hace toparnos con algunos nuevos problemas. A continuación, presentaremos las clases abstractas y las interfaces para tratar con esos problemas.

10.1 Simulaciones

Las computadoras se utilizan frecuentemente para simular sistemas reales. Entre estos se incluyen sistemas que modelan el flujo de tráfico en una ciudad, que permiten hacer predicciones meteorológicas, que simulan la propagación de una infección, que analizan el mercado bursátil, que llevan a cabo simulaciones medioambientales y muchos otros tipos. De hecho, muchas de las computadoras más potentes del mundo se utilizan para ejecutar algún tipo de simulación.

A la hora de crear una simulación por computadora, tratamos de modelar el comportamiento de un subconjunto del mundo real utilizando un modelo software. Toda simulación es, necesariamente, una simplificación de su modelo real. A menudo, decidir qué detalles dejar de lado y cuáles incluir en la simulación es enormemente complicado. Cuanto más detallada sea una simulación, más precisa será a la hora de predecir el comportamiento del sistema real.

Pero aumentar el grado de detalle hace que se incremente la complejidad del modelo y requiere aumentar tanto la potencia de procesamiento como el tiempo de programador necesario para desarrollar la aplicación. Un ejemplo bien conocido es el de la predicción meteorológica: los modelos climáticos en el campo de la meteorología se han ido incrementando añadiendo cada vez más detalles a lo largo de las últimas décadas. Como resultado, la precisión de las predicciones meteorológicas ha mejorado significativamente (aunque siguen estando lejos de ser perfectas, como todos hemos podido comprobar en alguna que otra ocasión). Buena parte de esta mejora ha sido posible gracias a los avances en la tecnología de computadoras.

La ventaja de las simulaciones es que podemos realizar experimentos que no podríamos llevar a cabo con el sistema real, bien porque no tenemos control sobre ese sistema (por ejemplo, el clima) o porque es demasiado costoso, demasiado peligroso o irreversible en caso de producirse un desastre. Podemos utilizar la simulación para investigar el comportamiento del sistema bajo ciertas circunstancias o podemos plantear cuestiones del tipo "¿Qué pasaría si...?".

Un ejemplo del uso de simulaciones medioambientales es el de intentar predecir los efectos de la actividad humana sobre los hábitats naturales. Considere el caso de un parque nacional en el que habitan una serie de especies protegidas e imagine que se plantea una propuesta para construir una autopista que atraviese el parque separándolo en sus dos mitades. Los partidarios de la autopista afirman que partir el parque por la mitad representa, en la práctica, una pérdida de terreno muy pequeña y no tiene ningún efecto sobre los animales que habitan en el parque, mientras que los ecologistas afirman lo contrario. ¿Cómo podemos decir cuál va a ser el efecto más probable, sin construir primero la autopista? Una opción posible es la simulación, aunque una cuestión esencial en todos los casos de este tipo será, por supuesto, "¿Qué nivel de precisión tiene esa simulación?" Uno puede "demostrar" lo que quiera con una simulación mal diseñada. Por ello, es esencial ir ganando confianza en la simulación a través de una serie de experimentos controlados.

La cuestión en este caso concreto se resume en si resulta importante para la supervivencia de una especie disponer de un hábitat conectado único o si dos áreas disjuntas (con la misma área total que la otra) servirían exactamente igual. En lugar de construir en primer lugar la autopista y luego observar lo que sucede, tratamos de simular el efecto con el fin de tomar una decisión racional.[1]

Nuestra simulación será necesariamente más simple que el escenario que acabamos de describir, porque lo estamos utilizando principalmente para ilustrar nuevas características del diseño y la implementación orientada a objetos. Por tanto, no tendrá la capacidad de simular con precisión muchos aspectos de la naturaleza, a pesar de lo cual algunas de las características de la simulación continúan siendo interesantes. En particular, permitirá ilustrar la estructura de las simulaciones típicas. Además, puede que le sorprenda su precisión. Sería un error suponer que una mayor complejidad nos da siempre una mayor precisión. A menudo se da el caso de que un modelo simplificado de algo puede proporcionar más información y permitir una mejor comprensión que otro modelo más complejo, en el que a menudo resulta más difícil aislar los mecanismos subyacentes –o incluso asegurarse de que el modelo es válido.

[1] En este caso particular, por cierto, el tamaño sí importa: el tamaño de un parque natural tiene un impacto significativo sobre su utilidad como hábitat para los animales.

10.2 La simulación de los zorros y los conejos

El escenario de simulación que hemos elegido para trabajar en este capítulo utiliza como base el ejemplo de la autopista que acabamos de mencionar. Implica observar las poblaciones de zorros y de conejos dentro de un área acotada. Esto es simplemente un ejemplo concreto de lo que se conoce como *simulaciones predador–presa*. Dichas simulaciones suelen utilizarse para modelar la variación en los tamaños de la población que resultan del hecho de que una especie predadora se alimente de otra especie que utiliza como presa. Existe un equilibrio delicado entres esas especies. Una gran población de presas puede proporcionar, potencialmente, una gran cantidad de alimento para una pequeña población de predadores. Sin embargo, demasiados predadores matarían a todas las presas, con lo que los cazadores se quedarían sin nada que comer. Los tamaños de las poblaciones también pueden verse afectados por el tamaño y la naturaleza del entorno. Por ejemplo, un entorno pequeño y cerrado podría conducir a la superpoblación y hacer que resultara sencillo para los predadores localizar sus presas, mientras que un entorno contaminado podría reducir la cantidad de presas disponibles e impedir que sobrevivieran ni siquiera una pequeña población de predadores. Puestos que los predadores, en un cierto contexto, suelen a menudo ser presas ellos mismos para otras especies (piense en los gatos, los pájaros y los gusanos, por ejemplo), la pérdida de una parte de la cadena alimenticia puede tener efectos muy importantes sobre la supervivencia de otras partes.

Como hemos hecho en los capítulos anteriores, partiremos de una versión de una aplicación que funciona perfectamente bien desde el punto de vista del usuario, pero cuyos aspectos internos no son tan buenos si los juzgamos según los principios de un buen diseño y una buena implementación orientadas a objetos. Utilizaremos esta versión básica para desarrollar varias versiones mejoradas que introducirán progresivamente nuevas técnicas de abstracción.

Un problema concreto que queremos abordar en la versión básica es que no hace un buen uso de las técnicas de herencia que presentamos en el Capítulo 8. Sin embargo, comenzaremos examinando el mecanismo de la simulación sin ser demasiado críticos con su implementación. Una vez que entendamos cómo funciona, estaremos en una buena posición para efectuar algunas mejoras.

Modelado de relaciones predador–presa Hay una larga tradición de intentos de modelado matemático de relaciones predador–presa antes de la invención de la computadora, porque ese tipo de modelos tiene importancia económica y no solo medioambiental. Por ejemplo, se utilizaron modelos matemáticos a principios del siglo XX para explicar las variaciones en la disponibilidad de bancos pesqueros en el mar Adriático, por el efecto de la Primera Guerra Mundial. Para encontrar más información acerca de la historia de este tema y entender quizá los conceptos de la dinámica de poblaciones, busque en la Web el modelo de Lotka-Volterra.

10.2.1 El proyecto *foxes-and-rabbits*

Abra el proyecto *foxes-and-rabbits-v1*. El diagrama de clases se muestra en la Figura 10.1.

Las principales clases en las que nos centraremos durante nuestras explicaciones son `Simulator`, `Fox` y `Rabbit`. Las clases `Fox` y `Rabbit` proporcionan modelos simples del comportamiento de un predador (zorro) y una presa (conejo), respectivamente. En esta

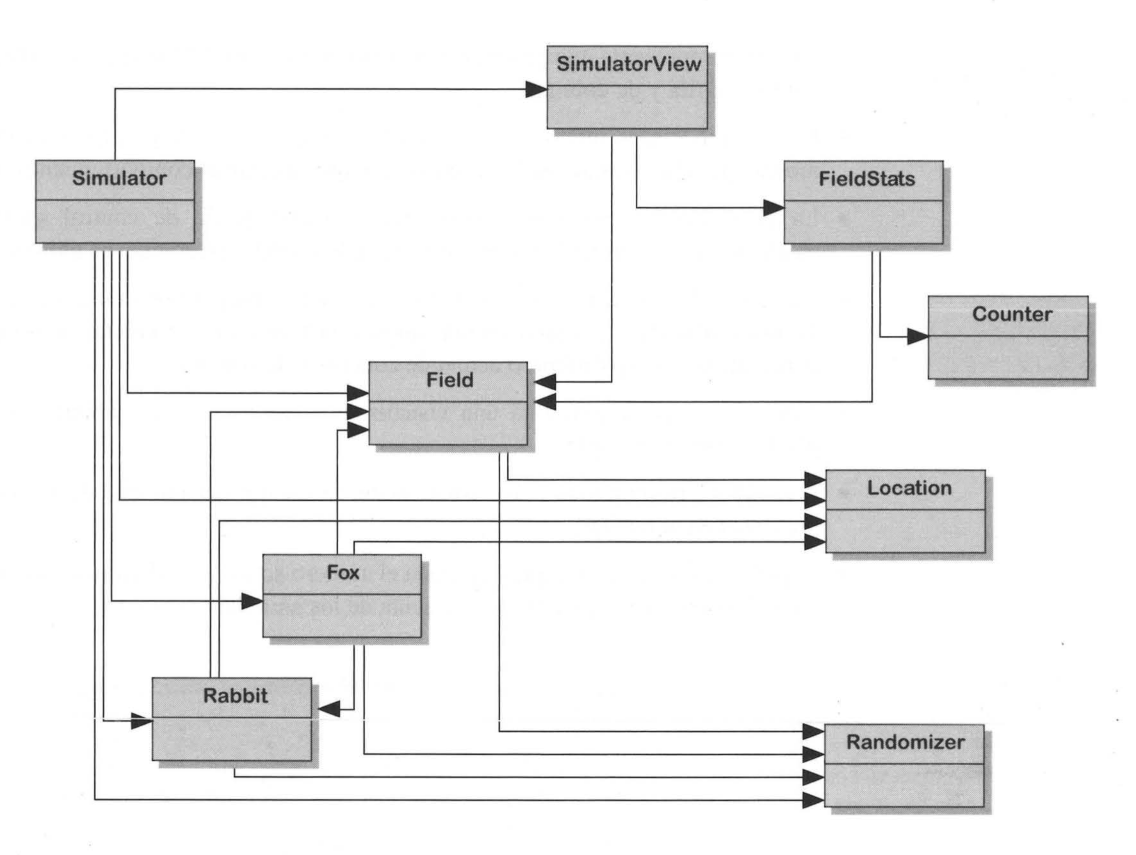


Figura 10.1 Diagrama de clases del proyecto *foxes-and-rabbits*.

implementación concreta no hemos intentado proporcionar un modelo biológico preciso de los zorros y los conejos reales, sino que simplemente tratamos de ilustrar los principios de las simulaciones predador–presa típicas. Nos centraremos principalmente en los aspectos que más afectan al tamaño de la población: nacimiento, muerte y suministro de alimentos.

La clase `Simulator` es responsable de crear el estado inicial de la simulación y luego ejecutar y controlar esta. La idea básica es muy simple: el simulador almacena colecciones de zorros y conejos, y da repetidamente a esos animales una oportunidad de vivir, ejecutando un paso[2] de su ciclo de vida. En cada paso, a cada zorro y a cada conejo se le permite llevar a cabo las acciones que caracterizan su comportamiento. Después de cada paso (cuando todos los animales han tenido la oportunidad de actuar), se muestra en pantalla el nuevo estado actual del hábitat.

Podemos resumir el propósito de las restantes clases de la forma siguiente:

- `Field` representa un hábitat bidimensional cerrado. El hábitat está compuesto por un número fijo de posiciones, que están dispuestas en filas y columnas. Como máximo, cada animal puede ocupar una única posición dentro del hábitat. Cada posición del hábitat puede albergar un animal o estar vacía.

[2] No vamos a definir cuánto tiempo representa en realidad cada "paso". En la práctica, esto tiene que decidirse mediante una combinación de cosas tales como qué es lo que estamos intentando descubrir, qué sucesos estamos simulando y cuánto tiempo real hay disponible para ejecutar la simulación.

- `Location` representa una posición bidimensional dentro del hábitat, especificada por sendos valores de fila y de columna.
- Estas cinco clases juntas (`Simulator`, `Fox`, `Rabbit`, `Field` y `Location`) proporcionan el modelo para la simulación. Determinan completamente el comportamiento de la misma.
- La clase `Randomizer` nos proporciona un cierto grado de control sobre los aspectos aleatorios de la simulación, como por ejemplo cuándo nacen nuevos animales.
- Las clases `SimulatorView`, `FieldStats` y `Counter` proporcionan una visualización gráfica de la simulación. La visualización muestra una imagen del hábitat, junto con contadores para cada especie (el número actual de conejos y de zorros).
- `SimulatorView` proporciona una visualización del estado del hábitat. En la Figura 10.2 puede verse un ejemplo.
- `FieldStats` proporciona a la visualización la cuenta del número de zorros y de conejos presentes en el hábitat.
- `Counter` es un contador que almacena el número actual de individuos para un cierto tipo de animal, como ayuda para llevar la cuenta de los animales existentes.

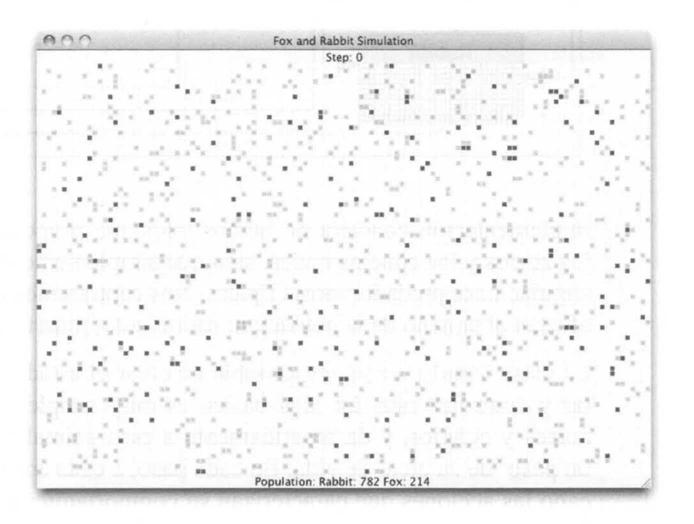


Figura 10.2
El estado inicial de la simulación *foxes-and-rabbits*.

Trate de hacer los siguientes ejercicios para entender cómo funciona la simulación antes de comenzar a leer las explicaciones acerca de su implementación.

Ejercicio 10.1 Cree un objeto `Simulator` utilizando el constructor sin parámetros; debería ver un estado inicial de la simulación similar al que se muestra en la Figura 10.2. Los rectángulos más numerosos representan los conejos. ¿Varía el número de zorros si invocamos una sola vez el método `simulateOneStep`, que ejecuta un paso de simulación?

Ejercicio 10.2 ¿Varía el número de zorros en cada paso? ¿Qué procesos naturales, de los que estamos modelando, cree que hacen que el número de zorros aumente o disminuya?

Ejercicio 10.3 Invoque el método `simulate` con un parámetro para ejecutar la simulación de modo continuo durante un número significativo de pasos, como por ejemplo 50 o 100. ¿Se incrementan o disminuyen al mismo ritmo el número de zorros y de conejos?

Ejercicio 10.4 ¿Qué cambios puede observar si ejecuta la simulación durante un periodo mucho más largo de tiempo, como por ejemplo 4.000 pasos? Puede utilizar el método `runLongSimulation` para hacer esto.

Ejercicio 10.5 Utilice el método `reset` para crear un nuevo estado inicial de la simulación y luego ejecútela de nuevo. ¿Se ejecuta esta vez una simulación idéntica? Si la respuesta es no, ¿observa si emergen de todos modos una serie de patrones similares?

Ejercicio 10.6 Si ejecuta una simulación durante el tiempo suficiente, ¿llegan a morir en algún momento todos los zorros o todos los conejos? En caso afirmativo, ¿podría señalar alguna razón por la que eso pueda estar ocurriendo?

Ejercicio 10.7 Anote el número de zorros y de conejos en cada uno de los primeros pasos de una simulación y al final de una simulación larga. Será útil disponer de un registro de esos valores cuando realicemos cambios posteriormente y llevemos a cabo pruebas de regresión.

Ejercicio 10.8 Después de haber estado ejecutando la simulación durante un tiempo, reiníciela y llame también al método estático `reset` de la clase `Randomizer`. Ahora ejecute de nuevo los primeros pasos; debería ver cómo se repite la simulación original. Examine el código de la clase `Randomizer` para ver si averigua la razón por la que esto sucede así. Puede que necesite examinar la API de la clase `java.util.Random` para responder a esta cuestión.

Ejercicio 10.9 Compruebe que si configuramos el campo `useShared` en `Randomizer` con el valor `false`, dejan de ser repetibles las simulaciones que hemos visto en el Ejercicio 10.8. Asegúrese de restaurar después el valor a `true`, porque la repetibilidad jugará un importante papel en las pruebas posteriores.

Ahora que tenemos una comprensión general y de carácter externo de lo que hace este proyecto, vamos a examinar en detalle la implementación de las clases `Rabbit`, `Fox` y `Simulator`.

10.2.2 La clase `Rabbit`

El código fuente de la clase `Rabbit` se muestra en el Código 10.1.

Código 10.1
La clase `Rabbit`.

```
// Se omiten las instrucciones import y el comentario de la clase

public class Rabbit
{
    // Características compartidas por todos los conejos
    // (variables de clase).

    // La edad a la que un conejo puede empezar a procrear.
    private static final int BREEDING_AGE = 5;
```

Código 10.1
(*continuación*)
La clase `Rabbit`.

```
    // La edad máxima hasta la que puede vivir un conejo.
    private static final int MAX_AGE = 40;
    // La probabilidad de que un conejo procree.
    private static final double BREEDING_PROBABILITY = 0.12;
    // El número máximo de nacimientos.
    private static final int MAX_LITTER_SIZE = 4;
    // Un generador de números aleatorios compartido para
    // controlar la reproducción.
    private static final Random rand = Randomizer.getRandom();

    // Características individuales (campos de instancia).

    // La edad del conejo.
    private int age;
    // Si el conejo está vivo o no.
    private boolean alive;
    // La posición del conejo.
    private Location location;
    // El hábitat ocupado.
    private Field field;

    /**
     * Crear un nuevo conejo. Puede crearse un conejo con edad
     * cero (un recién nacido) o con una edad aleatoria.
     *
     * @param randomAge Si es true, el conejo tendrá una edad
     *                  aleatoria.
     * @param field El hábitat actualmente ocupado.
     * @param location La posición dentro del hábitat.
     */
    public Rabbit(boolean randomAge, Field field, Location location)
    {
        // Se omite el cuerpo del constructor
    }

    /**
     * Esto es lo que el conejo hace la mayor parte del tiempo:
     * corre de un lado a otro. En ocasiones se reproducirá o
     * morirá por haber alcanzado la edad máxima.
     * @param newRabbits Una lista para devolver los conejos
     *                   recién nacidos.
     */
    public void run(List<Rabbit> newRabbits)
    {
        incrementAge();
        if(alive) {
            giveBirth(newRabbits);
```

Código 10.1
(continuación)
La clase `Rabbit`.

```
                // Tratar de moverse a una posición libre.
                Location newLocation =
                    field.freeAdjacentLocation(location);
                if(newLocation ! = null) {
                    setLocation(newLocation);
                }
                else {
                    // Superpoblación.
                    setDead();
                }
            }
        }

        /**
         * Indica que el conejo ya no está vivo.
         * Se le elimina del hábitat.
         */
        public void setDead()
        {
            alive = false;
            if(location != null) {
                field.clear(location);
                location = null;
                field = null;
            }
        }
        /**
         * Incrementar la edad.
         * Puede provocar la muerte del conejo.
         */
        private void incrementAge()
        {
            age++;
            if(age > MAX_AGE) {
                setDead();
            }
        }

        /**
         * Comprobar si este conejo va a dar a luz en este paso.
         * Los recién nacidos serán colocados en posiciones
         * adyacentes libres.
         *
         * @param newRabbits Una lista para devolver los conejos
         *                   recién nacidos.
         *
         */
```

Código 10.1
(continuación)
La clase `Rabbit`.

```
    private void giveBirth(List<Rabbit> newRabbits)
    {
        // Los conejos recién nacidos se colocan en
        // posiciones adyacentes.
        // Obtener una lista de las posiciones adyacentes libres.
     List<Location> free = field.getFreeAdjacentLocations(location);
        int births = breed();
        for(int b = 0; b < births && free.size() > 0; b++) {
            Location loc = free.remove(0);
            Rabbit young = new Rabbit(false, field, loc);
            newRabbits.add(young);
        }
    }

    /**
     * Si el conejo puede reproducirse, generar un número
     * que represente el número de nacimientos.
     * @return El número de nacimientos (puede ser cero).
     */
    private int breed()
    {
        int births = 0;
        if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY)
{
            births = rand.nextInt(MAX_LITTER_SIZE) + 1;
        }
        return births;
    }

    // otros métodos omitidos
}
```

La clase `Rabbit` contiene una serie de variables de clase que definen las opciones de configuración comunes a todos los conejos. Entre estas se incluyen los valores correspondientes al número máximo de años que un conejo puede vivir (definidos como un número de pasos de simulación) y el número máximo de crías que puede dar a luz en un paso cualquiera. El control centralizado de los aspectos aleatorios de la simulación se garantiza mediante un único objeto `Random` compartido suministrado por la clase `Randomizer`. Esto es lo que hace posible la repetibilidad que hemos visto en el Ejercicio 10.8. Además, cada conejo individual tiene cuatro variables de instancia que describen su estado: su edad expresada como número de pasos de simulación, si está todavía vivo y su posición dentro de un hábitat concreto.

Ejercicio 10.10 ¿Cree que omitir el género como atributo de la clase `Rabbit` puede producir una simulación imprecisa? Escriba los argumentos a favor y en contra de incluir el género.

Ejercicio 10.11 ¿Cree que nuestra implementación de la clase Rabbit incorpora otras simplificaciones, si comparamos el modelo con la vida real? Explique si esas simplificaciones pueden tener un impacto significativo sobre la precisión de la simulación.

Ejercicio 10.12 Experimente con los efectos de alterar algunos o todos los valores de las variables de clase de la clase Rabbit. Por ejemplo, ¿qué efecto tiene sobre las poblaciones que la probabilidad de procreación de los conejos sea mucho mayor o mucho menor de lo que es actualmente?

El comportamiento de un conejo se define en su método run, que a su vez utiliza los métodos giveBirth e incrementAge e implementa el movimiento del conejo. En cada paso de la simulación se invocará el método run y el conejo incrementará su edad. Si la edad es suficiente puede también reproducirse y después tratará de moverse. Tanto el comportamiento de movimiento como el de reproducción tienen componentes aleatorios. La dirección en la que se mueve el conejo se selecciona aleatoriamente, y la reproducción también tiene lugar de manera aleatoria, controlada por la variable de clase BREEDING_PROBABILITY.

Resultan obvias algunas de las simplificaciones que hemos hecho en nuestro modelo de los conejos: no hemos hecho ningún intento de distinguir entre machos y hembras, por ejemplo, y un conejo podría potencialmente reproducirse y alumbrar una nueva camada en cada paso de simulación, una vez que alcance la edad adecuada.

10.2.3 La clase Fox

Existe una gran similitud entre las clases Fox y Rabbit, por lo que en el Código 10.2 solo mostramos los elementos diferenciadores de Fox.

Código 10.2
La clase Fox.

```
// Se omiten las instrucciones import y el comentario de la clase

public class Fox
{
    // Características compartidas por todos los zorros
    // (variables de clase).

    // El valor alimenticio de un único conejo. En la práctica,
    // es el número de pasos de simulación que un zorro puede
    // ejecutar antes de tener que comer de nuevo.
    private static final int RABBIT_FOOD_VALUE = 9;

    // Se omiten otros campos estáticos

    // Características individuales (campos de instancia).

    // La edad del zorro.
    private int age;
    // Si el zorro está vivo o no.
    private boolean alive;
```

Código 10.2
(*continuación*)
La clase `Fox`.

```
    // La posición del zorro.
    private Location location;
    // El hábitat ocupado.
    private Field field;
    // El nivel alimenticio del zorro, que se incrementa
    // comiendo conejos.
    private int foodLevel;

    /**
     * Crear un zorro. Puede crearse un zorro como un recién
     * nacido (edad cero y no está hambriento) o con una edad y
     * un nivel alimenticio aleatorios.
     *
     * @param randomAge Si es true, el zorro tendrá una edad y un
     *                  nivel de hambre aleatorios.
     * @param field El hábitat actualmente ocupado.
     * @param location La posición dentro del hábitat.
     */
    public Fox(boolean randomAge, Field field, Location location)
    {
        // Se omite el cuerpo del constructor
    }

    /**
     * Esto es lo que hace el zorro la mayor parte del tiempo:
     * caza conejos. En el proceso, puede reproducirse, morir de
     * hambre o morir de viejo.
     * @param field El hábitat actualmente ocupado.
     * @param newFoxes Una lista para devolver los zorros
     *                 recién nacidos.
     *
     */
    public void hunt(List<Fox> newFoxes)
    {
        incrementAge();
        incrementHunger();
        if(alive) {
            giveBirth(newFoxes);
            // Moverse hacia una fuente de alimento
            // si la encuentra.
            Location newLocation = findFood();
            if(newLocation == null) {
                // No ha encontrado comida - tratar de moverse
                // a una posición libre.
                newLocation = field.freeAdjacentLocation(location);
            }
            // Ver si ha sido posible moverse.
            if(newLocation != null) {
```

Código 10.2
(continuación)
La clase Fox.

```
                    setLocation(newLocation);
                }
                else {
                    // Superpoblación.
                    setDead();
                }
            }
        }

        /**
         * Buscar conejos adyacentes a la posición actual.
         * Solo se come al primer conejo vivo.
         * @return Lugar en el que se encontró comida
         *         o null si no se ha encontrado.
         */
        private Location findFood()
        {
            List<Location> adjacent = field.adjacentLocations(location);
            Iterator<Location> it = adjacent.iterator();
            while(it.hasNext()) {
                Location where = it.next();
                Object animal = field.getObjectAt(where);
                if(animal instanceof Rabbit) {
                    Rabbit rabbit = (Rabbit) animal;
                    if(rabbit.isAlive()) {
                        rabbit.setDead();
                        foodLevel = RABBIT_FOOD_VALUE;
                        return where;
                    }
                }
            }
            return null;
        }

        // Se omiten los otros métodos
    }
```

Para los zorros, se invoca el método `hunt` en cada paso y es este el que define su comportamiento. Además de envejecer y posiblemente reproducirse en cada paso, el zorro busca comida (utilizando `findFood`). Si es capaz de encontrar un conejo en una posición adyacente, entonces mata al conejo y su nivel alimenticio se incrementa. Como sucede con los conejos, un zorro que no sea capaz de moverse se considerará muerto por la superpoblación.

Ejercicio 10.13 Como hizo para los conejos, evalúe el grado de simplificación existente en nuestro modelo de los zorros e indique si cree que esas simplificaciones pueden producir una simulación imprecisa.

Ejercicio 10.14 ¿Incrementar la edad máxima de los zorros conduce a un número significativamente mayor de zorros a lo largo de la simulación, o es más probable que la población de conejos se vea reducida a cero como resultado?

Ejercicio 10.15 Experimente con diferentes combinaciones de configuración (edad de reproducción, edad máxima, probabilidad de reproducción, tamaño de la camada, etc.) para los zorros y los conejos. ¿Desaparece siempre completamente alguna especie en ciertas configuraciones? ¿Hay configuraciones que sean estables, es decir, que produzcan un equilibrio de las poblaciones durante un intervalo de tiempo significativo?

Ejercicio 10.16 Experimente con diferentes tamaños de hábitat. (Puede hacer esto utilizando el segundo constructor de `Simulator`.) ¿Afecta el tamaño del hábitat a la probabilidad de supervivencia de las especies?

Ejercicio 10.17 Compare los resultados de ejecutar una simulación con un único hábitat de gran tamaño y dos simulaciones con hábitats que tengan la mitad del área del hábitat original. Esto modela una situación similar a la que se produciría si dividimos un área por la mitad construyendo una autopista. ¿Observa alguna diferencia significativa entre los dos escenarios, en lo que se refiere a la dinámica de las poblaciones?

Ejercicio 10.18 Repita las investigaciones del ejercicio anterior, pero varíe las proporciones de los dos hábitats más pequeños. Por ejemplo, divida el hábitat original en otros dos que tengan tres cuartos y un cuarto del área original, o dos tercios y un tercio. ¿Tiene alguna importancia el cómo dividamos el hábitat original?

Ejercicio 10.19 Actualmente, cada zorro come como máximo un conejo en cada paso. Modifique el método `findFood` para que todos los conejos situados en posiciones adyacentes sean comidos en un único paso. Evalúe el impacto de esta modificación sobre el resultado de la simulación. Observe que el método `findFood` devuelve actualmente la ubicación del único conejo devorado, por lo que en su versión tendrá que devolver la posición de uno de los conejos devorados. Sin embargo, no se olvide de devolver `null` si no hay ningún conejo que comer.

Ejercicio 10.20 Siguiendo con el ejercicio anterior, si un zorro se come varios conejos en un mismo paso, hay varias posibilidades distintas en lo que se refiere a cómo modelar su nivel alimenticio. Si sumamos los valores alimenticios de todos los conejos, el zorro tendrá un nivel alimenticio muy alto, haciendo que sea improbable que muera de hambre durante un largo periodo de tiempo. Alternativamente, podemos imponer un máximo al nivel alimenticio del zorro. Esto modela el efecto de un predador que mate a una presa independientemente de si tiene hambre o no. Evalúe en la simulación resultante el impacto de implementar esta opción.

Ejercicio 10.21 *Ejercicio avanzado* Dados los elementos aleatorios de la simulación, explique por qué los tamaños de las poblaciones en una simulación aparentemente estable podrían llegar a reducirse a cero.

10.2.4 La clase `Simulator`: configuración

La clase `Simulator` es el núcleo central de la aplicación, que coordina todas las restantes partes. El Código 10.3 ilustra algunas de sus características principales.

Código 10.3
Parte de la clase `Simulator`.

```
// Se omiten las instrucciones import y el comentario de la clase

public class Simulator
{
    // Omitidas las variables estáticas

    // Lista de animales en el hábitat.
    private List<Rabbit> rabbits;
    private List<Fox> foxes;
    // El estado actual del hábitat.
    private Field field;
    // El paso actual de la simulación.
    private int step;
    // Una vista gráfica de la simulación.
    private SimulatorView view;

    /**
     * Crear un hábitat de simulación con el tamaño indicado.
     * @param depth Profundidad del hábitat.
     *              Tiene que ser mayor que cero.
     * @param width Anchura del hábitat.
     *              Tiene que ser mayor que cero.
     */
    public Simulator(int depth, int width)
    {
        if(width <= 0 || depth <= 0) {
            System.out.println(
                 "The dimensions must be greater than zero.");
            System.out.println( "Using default values.");
            depth = DEFAULT_DEPTH;
            width = DEFAULT_WIDTH;
        }
        rabbits = new ArrayList<Rabbit>();
        foxes = new ArrayList<Fox>();
        field = new Field(depth, width);

        // Crear una vista del estado de cada posición
        // del hábitat.
        view = new SimulatorView(depth, width);
        view.setColor(Rabbit.class, Color.ORANGE);
        view.setColor(Fox.class, Color.BLUE);

        // Establecer un punto inicial válido.
        reset();
    }

    /**
     * Ejecutar la simulación durante el número de pasos
     * indicado a partir de su estado actual.
```

Código 10.3
(continuación)
Parte de la clase `Simulator`.

```
     * Detenerse antes del número de pasos indicado si
     * deja de ser viable.
     * @param numSteps El número de pasos que hay que ejecutar.
     */
    public void simulate(int numSteps)
    {
        for(int step = 1; step <= numSteps &&
                         view.isViable(field); step++) {
            simulateOneStep();
        }
    }

    /**
     * Ejecutar la simulación durante un único paso,
     * a partir de su estado actual.
     * Iterar para el hábitat, actualizando el estado de cada
     * zorro y de cada conejo.
     */
    public void simulateOneStep()
    {
        // Se omite el cuerpo del método
    }

    /**
     * Reinicializar la simulación a una posición de partida.
     */
    public void reset()
    {
        step = 0;
        rabbits.clear();
        foxes.clear();
        field.clear();
        populate();

        // Mostrar el estado inicial en la vista.
        view.showStatus(step, field);
    }

    /**
     * Poblar el hábitat con zorros y conejos.
     */
    private void populate()
    {
        Random rand = Randomizer.getRandom();
        field.clear();
        for(int row = 0; row < field.getDepth(); row++) {
            for(int col = 0; col < field.getWidth(); col++) {
```

Código 10.3 (*continuación*) Parte de la clase `Simulator`.

```
                    if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {
                        Location location = new Location(row, col);
                        Fox fox = new Fox(true, field, location);
                        foxes.add(fox);
                    }
                    else if(rand.nextDouble() <=
                                RABBIT_CREATION_PROBABILITY) {
                        Location location = new Location(row, col);
                        Rabbit rabbit = new Rabbit(true, field,
                                                   location);
                        rabbits.add(rabbit);
                    }
                    // En caso contrario, dejar la posición vacía.
                }
            }
        }
        // Se omiten los otros métodos
}
```

La clase `Simulator` tiene tres partes importantes: su constructor, el método `populate` y el método `simulateOneStep`. (El cuerpo de `simulateOneStep` se muestra más adelante.)

Cuando se crea un objeto `Simulator`, este se encarga de construir todas las restantes partes de la simulación (el hábitat, las listas para almacenar los distintos tipos de animales y la interfaz gráfica). Una vez configurados todos estos aspectos, se invoca (indirectamente, a través del método `reset`) el método `populate` del simulador, para crear las poblaciones iniciales. Se utilizan diferentes probabilidades para decidir si una posición contendrá a uno de estos animales. Observe que a los animales que se crean al principio de la simulación se les asigna una edad inicial aleatoria. Esto tiene dos objetivos:

- Representa de forma más precisa una población con edades diversas, que debería ser el estado normal de la simulación.
- Si todos los animales comenzarán con una edad igual a cero, no se crearían nuevos animales hasta que la población inicial hubiera alcanzado su edad de reproducción correspondiente. Puesto que los zorros devoran a los conejos independientemente de la edad del zorro, hay un riesgo de que la población de conejos sea exterminada antes de tener la posibilidad de reproducirse o de que la población de zorros muera de hambre.

Ejercicio 10.22 Modifique el método `populate` de `Simulator` para determinar si el fijar una edad inicial igual a cero para los zorros y los conejos es siempre catastrófico. Asegúrese de ejecutar la simulación el suficiente número de veces –¡con diferentes estados iniciales, por supuesto!

Ejercicio 10.23 Si se selecciona una edad inicial aleatoria para los conejos pero no para los zorros, la población de conejos tenderá a hacerse muy grande, mientras que la población de zorros seguirá siendo pequeña. Una vez que los zorros tengan la edad suficiente para reproducirse, ¿tiende la simulación a comportarse de nuevo como la versión original? ¿Qué es lo que esto sugiere acerca de los tamaños relativos de las poblaciones iniciales y su impacto sobre el resultado de la simulación?

10.2.5 La clase Simulator: un paso de simulación

La parte central de la clase Simulator es el método simulateOneStep mostrado en el Código 10.4. Utiliza bucles separados para dejar que cada tipo de animal se mueva (y posiblemente se reproduzca o haga lo que esté programado para hacer). Dado que cada animal puede alumbrar a nuevos animales, a los métodos hunt y run de Fox y Rabbit se les pasan como parámetros listas donde almacenar los animales recién nacidos. Después, al final de cada paso, los animales recién nacidos se añaden a las listas maestras. Ejecutar simulaciones más largas resulta trivial: para hacerlo, se invoca repetidamente el método simulateOneStep en un bucle simple.

Para dejar que cada animal actúe, el simulador mantiene listas separadas de los diferentes tipos de animales. Aquí, no hacemos ningún uso de la herencia y la situación nos recuerda a la primera versión del proyecto *network* presentado en el Capítulo 8.

Código 10.4
Dentro de la clase Simulator: simulación de un único paso.

```
public void simulateOneStep()
{
    step++;
    // Proporcionar espacio para los conejos recién nacidos.
    List<Rabbit> newRabbits = new ArrayList<Rabbit>();
    // Dejar que actúen todos los conejos.
    for(Iterator<Rabbit> it = rabbits.iterator(); it.hasNext(); ) {
        Rabbit rabbit = it.next();
        rabbit.run(newRabbits);
        if(!rabbit.isAlive()) {
            it.remove();
        }
    }

    // Proporcionar espacio para los zorros recién nacidos.
    List<Fox> newFoxes = new ArrayList<Fox> ();
    // Dejar que actúen todos los zorros.
    for(Iterator<Fox> it = foxes.iterator(); it.hasNext(); ) {
        Fox fox = it.next();
        fox.hunt(newFoxes);
        if(!fox.isAlive()) {
            it.remove();
        }
    }


    // Añadir los conejos y zorros recién nacidos a las
    // listas principales.
    rabbits.addAll(newRabbits);
    foxes.addAll(newFoxes);
    view.showStatus(step, field);
}
```

Ejercicio 10.24 Cada animal está siempre almacenado en dos estructuras de datos distintas: el hábitat `Field` y las listas `rabbits` y `foxes` de `Simulator`. Existe el riesgo de que esas estructuras sean incoherentes entre sí. Asegúrese de que comprende perfectamente cómo se mantiene la coherencia entre `Field` y las listas de animales, gracias a la acción del método `simulateOneStep` de `Simulator`, de `hunt` en `Fox` y de `run` en `Rabbit`.

Ejercicio 10.25 ¿Cree que sería mejor que `Simulator`, en lugar de mantener listas separadas de zorros y conejos, generara esas listas de nuevo a partir del contenido del hábitat, al principio de cada paso de simulación? Explique su respuesta.

Ejercicio 10.26 Escriba una prueba para garantizar que, al final de cada paso de simulación, no haya ningún animal (muerto o vivo) en el hábitat que no se encuentre dentro de las listas y viceversa. ¿Debería haber algún animal muerto en cualquiera de esos lugares al llegar a ese punto?

10.2.6 Intentos de mejora de la simulación

Ahora que hemos examinado como funciona la simulación, estamos en disposición de efectuar mejoras en su diseño interno y en su implementación. En las siguientes secciones nos vamos a centrar en la realización de mejoras progresivas, mediante la introducción de nuevas características de programación. Hay varios puntos por los que podríamos empezar, pero una de las debilidades más obvias es que no se ha hecho ningún intento de aprovechar las ventajas de la herencia en las clases `Fox` y `Rabbit`, que comparten una gran cantidad de elementos comunes. Para hacer esto, vamos a introducir el concepto de *clase abstracta.*

Ejercicio 10.27 Identifique las similitudes y diferencias entre las clases `Fox` y `Rabbit`. Haga listas separadas de los campos, métodos y constructores, distinguiendo entre las variables de clase (campos estáticos) y las variables de instancia.

Ejercicio 10.28 Los métodos candidatos a ser incluidos en una superclase son aquellos que sean idénticos en todas las subclases. ¿Qué métodos son verdaderamente idénticos en las clases `Fox` y `Rabbit`? Para llegar a una conclusión, puede tratar de considerar el efecto de sustituir los valores de las variables de clase en los cuerpos de los métodos que las utilizan.

Ejercicio 10.29 En la versión actual de la simulación, los valores de todas las variables de clase de nombre similar son distintos. Si los dos valores de una variable de clase concreta (`BREEDING_AGE`, por ejemplo) fueran idénticos, ¿haría eso variar sus conclusiones acerca de qué métodos son verdaderamente idénticos?

10.3 Clases abstractas

En el Capítulo 8 hemos presentado conceptos tales como la herencia y el polimorfismo que deberíamos ser capaces de aprovechar en la aplicación de simulación. Por ejemplo, las clases `Fox` y `Rabbit` comparten muchas características similares, lo que sugiere que deberían ser subclases de una superclase común, como por ejemplo `Animal`. En esta sección vamos a comenzar a realizar esas modificaciones, con el fin de mejorar el diseño y la implementación

de la simulación como un todo. Como sucede con el proyecto del Capítulo 8, la utilización de una superclase común debería evitarnos duplicar el código en las subclases y debería permitirnos también simplificar el código en la clase cliente (que aquí es `Simulator`). Es importante recalcar que estamos emprendiendo un proceso de refactorización y que esas modificaciones no deberían variar las características esenciales de la simulación, tal como se percibe desde el punto de vista de un usuario.

10.3.1 La superclase `Animal`

Para el primer conjunto de cambios, vamos a mover los elementos idénticos de `Fox` y `Rabbit` a una superclase `Animal`. El proyecto *foxes-and-rabbits-v1* proporciona una copia de la versión base de la simulación, para que pueda seguir las modificaciones que vayamos realizando.

- Tanto `Fox` como `Rabbit` definen los atributos `age`, `alive`, `field` y `location`. Sin embargo, en este punto solo vamos a mover `alive`, `location` y `field` a la superclase `Animal` y veremos posteriormente qué hacer con el campo `age`. Tal como solemos hacer con los campos de instancia, mantendremos todos ellos como privados en la superclase. Los valores iniciales se configuran en el constructor de `Animal`, asignando a `alive` el valor `true`, y pasándose `field` y `location` mediante llamadas `super` desde los constructores de `Fox` y de `Rabbit`.
- Estos campos necesitarán métodos selectores y mutadores, así que podemos mover los métodos `getLocation`, `setLocation`, `isAlive` y `setDead` existentes de `Fox` y `Rabbit`. También necesitaremos añadir un método `getField` en `Animal`, para poder sustituir el acceso directo a `field` desde los métodos `run`, `hunt`, `giveBirth` y `findFood` de la subclase.
- Al mover estos métodos, tenemos que pensar en cuál es la visibilidad más adecuada para los mismos. Por ejemplo, `setLocation` es privado tanto en `Fox` como en `Rabbit`, pero no puede mantenerse privado en `Animal`, porque entonces `Fox` y `Rabbit` no podrían invocarlo. Por tanto, debemos ascenderlo al nivel de visibilidad `protected`, para indicar que es un método pensado para que las subclases lo invoquen.
- De forma similar, observe que `setDead` era público en `Rabbit` pero privado en `Fox`. ¿Debería entonces ser público en `Animal`? Era público en `Rabbit` porque los zorros necesitaban poder invocar el método `setDead` de un conejo al comerse su presa. Ahora que son clases hermanas de una superclase compartida, una visibilidad más apropiada será `protected`, indicando de nuevo que se trata de un método que no es parte de la interfaz general de un animal, al menos en esta etapa de desarrollo del proyecto.

La realización de estas modificaciones constituye un primer paso hacia la eliminación del código duplicado mediante el uso de la herencia, de forma bastante similar a como lo hicimos en el Capítulo 8.

Ejercicio 10.30 ¿Qué tipo de estrategia de pruebas de regresión podría establecerse antes de acometer la tarea de refactorizar la simulación? ¿Son unas pruebas que podrían automatizarse fácilmente?

Ejercicio 10.31 La clase `Randomizer` nos proporciona una forma de controlar si los elementos "aleatorios" de la simulación son repetibles o no. Si se asigna el valor `true` a su campo `useShared`, entonces todos los objetos de la simulación comparten un mismo objeto `Random`. Además, su método `reset` reinicializa el

punto de partida del objeto `Random` compartido. Utilice estas características mientras resuelve el siguiente ejercicio, para verificar que no está modificando ningún aspecto fundamental de la simulación global al introducir una clase `Animal`.

Cree la superclase `Animal` en su versión del proyecto. Efectúe las modificaciones explicadas anteriormente. Asegúrese de que la simulación funciona de forma similar a como lo hacía antes. Puede comprobar esto teniendo la versión antigua del proyecto y la nueva abiertas a la vez, por ejemplo, y realizando llamadas idénticas a los objetos `Simulator` en ambas versiones, para verificar después que los resultados son idénticos.

Ejercicio 10.32 ¿Cómo ha mejorado la utilización de la herencia el proyecto hasta el momento? Explique su respuesta.

10.3.2 Métodos abstractos

Hasta aquí, el uso de la clase `Animal` nos ha ayudado a evitar buena parte de la duplicación de código en las clases `Rabbit` y `Fox`, y ha hecho que sea potencialmente más fácil añadir nuevos tipos de animales en el futuro. Sin embargo, como hemos visto en el Capítulo 8, el uso inteligente de la herencia debería simplificar también la clase cliente, en este caso, `Simulator`. Vamos a investigar ahora este aspecto.

En la clase `Simulator`, hemos utilizado listas de zorros y conejos con tipos de objeto diferentes, hemos empleado también un código de iteración específico de cada lista para implementar cada paso de la simulación. El Código 10.4 muestra el código relevante. Ahora que tenemos la clase `Animal`, podemos mejorar ese fragmento de código. Puesto que todos los objetos de nuestras colecciones de animales son ahora un subtipo de `Animal`, podemos combinarlos en una única colección e iterar así una única vez utilizando el tipo `Animal`. Sin embargo, es evidente que surge un problema en la solución con una única lista mostrada en el Código 10.5. Aunque sabemos que cada elemento de la lista es un elemento `Animal`, seguimos teniendo que averiguar de qué tipo de animal se trata para poder invocar el método de acción correcto para ese tipo –`run` o `hunt`. Determinamos el tipo mediante el operador `instanceof`.

Código 10.5
Una solución con una única lista, que resulta poco satisfactoria a la hora de hacer que los animales actúen.

```
for(Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {
    Animal animal = it.next();
    if(animal instanceof Rabbit) {
        Rabbit rabbit = (Rabbit) animal;
        rabbit.run(newAnimals);
    }
    else if(animal instanceof Fox) {
        Fox fox = (Fox) animal;
        fox.hunt(newAnimals);
    }
    else {
        System.out.println("found unknown animal");
    }
    // Eliminar a los animales muertos de la simulación.
    if(! animal.isAlive()) {
        it.remove();
    }
}
```

El hecho de que en el Código 10.5 haya que comprobar independientemente cada tipo de animal y cualificar su tipo mediante un cast, y el hecho de que exista código especial para cada clase de animal, son buenos indicios de que no estamos aprovechando por completo las ventajas que la herencia nos ofrece. Una solución mejor consiste en incluir un método en la superclase (`Animal`), permitiendo a un animal actuar y luego sustituir ese método en cada subclase, de modo que dispongamos de una llamada a método polimórfica, que permita a cada animal actuar apropiadamente sin necesidad de comprobar cada tipo específico de animal. Esta es una técnica habitual de refactorización en situaciones como esta, en las que se invoca un comportamiento específico del subtipo desde un contexto que solo trata con el supertipo.

Vamos a suponer que creamos un método como ese denominado `act`, y vamos a analizar el código fuente resultante. El Código 10.6 muestra la implementación de esta solución.

Código 10.6
La solución mejorada del código que implementa la acción de los animales.

```
// Dejar que todos los animales actúen.
for(Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {
    Animal animal = it.next();
    animal.act(newAnimals);
    // Eliminar los animales muertos de la simulación.
    if(! animal.isAlive()) {
      it.remove();
    }
}
```

Son varias las observaciones importantes que podemos hacer:

- La variable que estamos utilizando para cada elemento de la colección (`animal`) es de tipo `Animal`. Esto es legal, porque todos los objetos de la colección son zorros o conejos y todos son subtipos de `Animal`.
- Asumimos que los métodos específicos de acción (`run` para `Rabbit`, `hunt` para `Fox`) han sido renombrados como `act`. Esto es más apropiado que la solución anterior. En lugar de decirle a cada animal exactamente lo que tiene que hacer, nos limitamos a decirle que "actúe" (*act*) y dejamos que sea el propio animal el que decida qué es exactamente lo que quiere hacer. Esto reduce el acoplamiento entre `Simulator` y las subclases individuales de animales.
- Puesto que el tipo dinámico de la variable determina qué método llegará realmente a ejecutarse (como hemos explicado en el Capítulo 9), se ejecutará para los zorros el método de acción correspondiente a los zorros y el método de acción correspondiente a los conejos se ejecutará para los conejos.
- Puesto que la comprobación de tipos se efectúa usando el tipo estático, este código solo podrá compilarse si la clase `Animal` dispone de un método `act` con la cabecera correcta.

El único problema que queda es precisamente el último de estos puntos. Dado que estamos usando la instrucción:

```
animal.act(newAnimals);
```

y la variable `animal` es de tipo `Animal`, esto solo podrá compilarse si `Animal` define un método `act`, como hemos visto en el Capítulo 9. Sin embargo, la situación aquí es bastante distinta de aquella con la que nos encontramos al analizar el método `display` en el Capítulo 9. Allí, la

versión de `display` en la superclase sí que tenía un trabajo útil que hacer: imprimir los campos definidos en la superclase. Aquí, aunque cada animal concreto tiene un conjunto específico de acciones que realizar, no podemos describir con ningún grado de detalle las acciones que los animales en general realicen. Las acciones concretas dependen de cada subtipo específico.

Nuestro problema consiste en decidir cómo deberíamos definir el método `act` de `Animal`.

El problema refleja el hecho de que jamás va a existir ninguna instancia de la clase `Animal`. No hay ningún objeto dentro de nuestra simulación (ni tampoco en la Naturaleza) que sea simplemente un animal y que no sea a la vez una instancia de una subclase más específica. Estos tipos de clases, que no están pensadas para crear objetos sino solo para servir como superclases, se conocen con el nombre de *clases abstractas*. Por ejemplo, para los animales podemos decir que cada animal puede actuar, pero no es posible describir exactamente cómo actuará sin hacer referencia a una subclase más específica. Esto es algo típico de las clases abstractas y tiene su reflejo en las estructuras Java correspondientes.

Concepto:

La definición de un **método abstracto** está compuesta de una cabecera de método, sin que exista un cuerpo del mismo. Se marca con la palabra clave `abstract`.

Para la clase `Animal`, queremos indicar que cada animal dispone de un método `act`, pero no podemos proporcionar una implementación razonable en la clase `Animal`. La solución en Java consiste en declarar el método *abstracto*. He aquí un ejemplo de un método `act` abstracto:

```
abstract public void act(List<Animal> newAnimals);
```

Un método abstracto se caracteriza por dos detalles:

- Utiliza como prefijo la palabra clave `abstract`.
- No tiene un cuerpo del método. En lugar de ello, su cabecera se termina con un punto y coma.

Puesto que el método no tiene cuerpo, nunca se puede ejecutar. Pero, como ya hemos dicho, no queremos ejecutar el método `act` de un objeto `Animal`, por lo que eso no constituye un problema.

Antes de ponernos a investigar en detalle los efectos de utilizar un método abstracto, vamos a presentar de manera más formal el concepto de clase abstracta.

10.3.3 Clases abstractas

Concepto:

Una **clase abstracta** es una clase que no está pensada para crear instancias. Su objetivo es servir como una superclase de otras clases. Las clases abstractas pueden contener métodos abstractos.

No solo los métodos pueden declararse como abstractos, sino que también se pueden declarar como abstractas las clases. El Código 10.7 muestra un ejemplo de la clase `Animal` como clase abstracta. Las clases se declaran como abstractas insertando la palabra `abstract` en la cabecera de la clase.

Las clases que no son abstractas (todas las que hemos visto anteriormente) se denominan *clases concretas*.

La declaración de una clase como abstracta tiene varios objetivos:

- No pueden crearse instancias de las clases abstractas. Tratar de utilizar la palabra clave `new` con una clase abstracta es un error y el compilador no lo permitirá. Esto tiene su correspondencia dentro de BlueJ: si hacemos clic con el botón derecho del ratón sobre una clase abstracta en el diagrama de clases, no aparecerá ningún constructor en el menú emergente. Esto cumple perfectamente con el objetivo que nos habíamos marcado: como hemos dicho, no queremos crear directamente instancias de la clase `Animal` –esta clase solo sirve como superclase. Declarar una clase como abstracta obliga a que siempre se cumpla esta restricción.

Código 10.7
`Animal` como una clase abstracta.

```
public abstract class Animal
{
    // Campos omitidos

    /**
     * Hacer que este animal actúe: es decir, hacer que lleve a
     * cabo lo que quiera/necesite hacer.
     * @param newAnimals Una lista para devolver los animales
     *                  recién nacidos.
     */
    abstract public void act(List<Animal> newAnimals);

    // Otros métodos omitidos
}
```

Concepto:

Subclase abstracta Para que una subclase de una clase abstracta se transforme en concreta, debe proporcionar implementaciones para todos los métodos abstractos heredados. En caso contrario, la propia subclase será también abstracta.

- Solo las clases abstractas pueden disponer de métodos abstractos. Esto garantiza que todos los métodos de las clases concretas puedan siempre ejecutarse. Si permitimos que exista un método abstracto de una clase concreta, entonces sería posible crear una instancia de una clase que carezca de la implementación correspondiente a un método.
- Las clases abstractas con métodos abstractos obligan a las subclases a sustituir e implementar esos métodos declarados como abstractos. Si una subclase no proporciona una implementación para un método abstracto heredado, será abstracta ella misma, y no se podrá crear ninguna instancia. Para que una subclase sea concreta, debe proporcionar implementaciones para *todos* los métodos abstractos heredados.

Ahora podemos comenzar a ver cuál es el propósito de los métodos abstractos. Aunque no proporcionan una implementación, garantizan de todos modos que todas las subclases concretas dispongan de una implementación de dicho método. En otras palabras, aun cuando la clase `Animal` no implementa el método `act`, garantiza que todos los animales existentes dispongan de un método `act` implementado. Esto se lleva a cabo garantizando que:

- no se pueda crear directamente ninguna instancia de la clase `Animal` y
- todas las subclases concretas estén obligadas a implementar el método `act`.

Aunque no podemos crear directamente una instancia de una clase abstracta, por lo demás podemos emplear una clase abstracta como tipo en las formas habituales. Por ejemplo, las reglas normales del polimorfismo nos permiten gestionar los zorros y los conejos como instancias de la clase `Animal`. Por tanto, aquellas partes de la simulación que no necesiten saber si están tratando con una subclase específica podrán usar en su lugar el tipo correspondiente a la superclase.

Ejercicio 10.33 Aunque el cuerpo del bucle en el Código 10.6 ya no maneja los tipos `Fox` y `Rabbit`, sigue manejando el tipo `Animal`. ¿Por qué no es posible tratar cada objeto de la colección simplemente empleando el tipo `Object`?

Ejercicio 10.34 Si una clase dispone de uno o más métodos abstractos, ¿es necesario definirla como abstracta? Si no está seguro, experimente con el código de la clase `Animal` en el proyecto *foxes-and-rabbits-v2*.

Ejercicio 10.35 Si una clase no tiene ningún método abstracto, ¿se la puede definir como abstracta? Si no está seguro, modifique `act` para que sea un método concreto dentro de la clase `Animal`, dotándole de un cuerpo de método sin ninguna instrucción.

Ejercicio 10.36 ¿Podría alguna vez tener sentido definir una clase como abstracta si no tiene ningún método abstracto? Explique su respuesta.

Ejercicio 10.37 ¿Qué clases del paquete `java.util` son abstractas? Algunas de ellas contienen la palabra `Abstract` en el nombre de la clase, pero ¿existe alguna otra forma de deducirlo a partir de la documentación? ¿Qué clases concretas amplían las definiciones de esas clases abstractas?

Ejercicio 10.38 ¿Se puede decir, a partir de la documentación de la API de una clase abstracta, cuáles de sus métodos son abstractos (si es que hay alguno)? ¿Es *necesario* saber qué métodos son abstractos?

Ejercicio 10.39 Revise las reglas de sustitución para métodos y campos vistas en el Capítulo 9. ¿Por qué son particularmente importantes en nuestros intentos por introducir la herencia en esta aplicación?

Ejercicio 10.40 Las modificaciones realizadas en esta sección han eliminado las dependencias (acoplamientos) del método `simulateOneStep` con respecto a las clases `Fox` y `Rabbit`. Sin embargo, la clase `Simulator` sigue estando acoplada con `Fox` y `Rabbit`, porque el método `populate` hace referencia a esas clases. No hay ninguna forma de evitar esto; cuando creamos instancias de animales tenemos que especificar exactamente qué tipo de animal queremos crear.

Podríamos mejorar esto dividiendo `Simulator` en dos clases: una clase, `Simulator`, que se encargue de ejecutar la simulación que esté completamente desacoplada de las clases de animales concretos; la otra clase, `PopulationGenerator` (creada e invocada por el simulador), se encargaría de crear la población. Solo esta clase estará acoplada a las clases de animales concretos, lo que facilita al programador de mantenimiento localizar los lugares donde es necesario efectuar cambios cuando se amplía la aplicación. Trate de implementar esta tarea de refactorización. La clase `PopulationGenerator` debería también definir los colores para cada tipo de animal.

El proyecto *foxes-and-rabbits-v2* proporciona una implementación de nuestra simulación con las mejoras que hemos explicado aquí. Es importante resaltar que la modificación en `Simulator` para procesar todos los animales mediante una única lista, en lugar de emplear listas separadas, implica que los resultados de la simulación en la versión 2 no serán idénticos a los de la versión 1.

En los proyectos del libro, podrá encontrar una tercera versión de este proyecto: *foxes-and-rabbits-graph*. Este proyecto es idéntico a *foxes-and-rabbits-v2* en lo que respecta a su modelo (es decir, las implementaciones de animales/zorros/conejos/simulador), pero añade una segunda vista al proyecto: una gráfica que muestra el tamaño de las poblaciones a lo largo del tiempo. Hablaremos de algunos aspectos de su implementación un poco más adelante en este mismo capítulo. Por ahora, vamos a limitarnos a experimentar con este proyecto.

Ejercicio 10.41 Abra y ejecute el proyecto *foxes-and-rabbits-graph*. Preste atención a la salida *Graph View*. Explique por escrito el significado de la gráfica e intente razonar por qué tiene el aspecto que tiene. ¿Existe alguna relación entre las dos curvas?

Ejercicio 10.42 Repita algunos de sus experimentos con diferentes tamaños de hábitats (especialmente con hábitats más pequeños). ¿Proporciona la vista de la gráfica alguna nueva información significativa, o le ayuda a comprender o explicar lo que ve?

Si ha hecho todos los ejercicios de este capítulo hasta el momento, entonces su versión del proyecto será la misma que *foxes-and-rabbits-v2* y similar a *foxes-and-rabbits-graph*, excepto por la visualización de la gráfica. Puede continuar realizando los ejercicios del resto del capítulo con cualquiera de las dos versiones de este proyecto.

10.4 Más métodos abstractos

Cuando creamos la superclase `Animal` en la Sección 10.3, lo hicimos identificado los elementos comunes de las subclases, pero decidimos no mover el campo `age` ni los métodos asociados con el mismo. Esta actitud quizá sea demasiado conservadora. Podríamos, de hecho, haber movido con bastante facilidad el campo `age` a `Animal` y haber proporcionado allí un método selector y otros mutadores que pudieran ser invocados por los métodos de las subclases, como por ejemplo `incrementAge`. Entonces, ¿por qué no hemos movido `incrementAge` y `canBreed` a `Animal`? La razón para no mover esos métodos es que, aunque varios de los cuerpos de los métodos restantes en `Fox` y `Rabbit` contienen instrucciones textualmente idénticas, su uso de variables de clase con diferentes valores implica que no pueden moverse directamente a la superclase. En el caso de `canBreed`, el problema es la variable `BREEDING_AGE`, mientras que `breed` depende de `BREEDING_PROBABILITY` y `MAX_LITTER_SIZE`. Por ejemplo, si moviéramos `canBreed` a `Animal`, entonces el compilador necesitaría poder acceder en la clase `Animal` a un valor para la edad de reproducción específica del subtipo. Es tentador definir un campo `BREEDING_AGE` en la clase `Animal` y asumir que su valor será sustituido por otros campos denominados de la misma manera en las subclases. Sin embargo, los campos en Java se gestionan de forma distinta a los métodos: no pueden ser sustituidos por versiones específicas de las subclases.[3] Esto significa que un método `canBreed` en `Animal` utilizaría un valor sin sentido definido dentro de esa clase, en lugar de emplear el valor específico de una subclase determinada.

El hecho de que el valor del campo no tuviera sentido nos proporciona una pista de cómo podemos resolver este problema y, como resultado, mover desde las subclases a la superclase una mayor cantidad de métodos similares.

Recuerde que hemos definido `act` como abstracto en `Animal` porque no tendría sentido disponer de un cuerpo para el método. Si accedemos a la edad de reproducción mediante un método, en lugar de mediante un campo, podemos resolver los problemas asociados con las propiedades dependientes de la edad. Este enfoque se ilustra en el Código 10.8.

El método `canBreed` se ha movido a `Animal` y se ha vuelto a escribir para que utilice el valor devuelto por una llamada a método, en lugar de emplear el valor de una variable de clase. Para que esto funcione, habrá que definir un método `getBreedingAge` en la clase `Animal`. Puesto que no podemos especificar una edad de reproducción para los animales en general,

[3] Esta regla se aplica independientemente de si un campo es estático o no.

Código 10.8
El método `canBreed` de `Animal`.

```
/**
 * Un animal puede reproducirse si ha alcanzado la
 * edad de reproducción.
 * @return true si el animal puede reproducirse
 */
public boolean canBreed()
{
    return age >= getBreedingAge();
}

/**
 * Devolver la edad de reproducción de este animal.
 * @return La edad de reproducción de este animal.
 */
abstract protected int getBreedingAge();
```

Concepto:

Llamadas a métodos en la superclase Las llamadas a métodos de instancia no privados desde dentro de una superclase siempre se evalúan en el contexto más amplio del tipo dinámico del objeto.

podemos de nuevo emplear un método `abstract` en la clase `Animal` y hacer uso de una serie de redefiniciones concretas en las subclases. Tanto `Fox` como `Rabbit` definirán sus propias versiones de `getBreedingAge` para devolver sus valores concretos de `BREEDING_AGE`:

```
/**
  * @return La edad a la que un conejo comienza a reproducirse.
  */
public int getBreedingAge()
{
    return BREEDING_AGE;
}
```

Por tanto, aunque la llamada a `getBreedingAge` se origina en el código de la superclase, el método invocado se define en la subclase. Esto puede parecer algo misterioso a primera vista, pero está basado en los mismos principios que hemos descrito en el Capítulo 9, en el sentido de que se emplea el tipo dinámico de un objeto para determinar qué versión de un método se invoca en tiempo de ejecución. La técnica ilustrada aquí hace posible que cada instancia utilice el valor apropiado para su tipo de subclase. Utilizando la misma técnica, podemos mover los métodos restantes, `incrementAge` y `breed`, a la superclase.

Ejercicio 10.43 Utilizando su última versión del proyecto (o el proyecto *foxes-and-rabbits-v2* si no ha hecho todos los ejercicios), anote el número de zorros y de conejos a lo largo de un pequeño número de pasos para preparase para las pruebas de regresión aplicables a los cambios que vamos a realizar.

Ejercicio 10.44 Mueva el campo `age` de `Fox` y `Rabbit` a `Animal`. Inicialícelo a cero en el constructor. Proporcione métodos selector y mutador para ese campo y emplee los métodos en `Fox` y `Rabbit` en lugar de usar accesos directos al campo. Asegúrese de que el programa se compila y ejecuta como antes.

Ejercicio 10.45 Mueva el método `canBreed` de `Fox` y `Rabbit` a `Animal`, escríbalo de nuevo como se muestra en el Código 10.8. Proporcione versiones apropiadas de `getBreedingAge` en `Fox` y `Rabbit` que devuelvan los diferentes valores de la edad de reproducción.

Ejercicio 10.46 Mueva el método `incrementAge` de `Fox` y `Rabbit` a `Animal` proporcionando un método abstracto `getMaxAge` en `Animal`, junto con sus correspondientes versiones concretas en `Fox` y `Rabbit`.

Ejercicio 10.47 ¿Puede moverse el método `breed` a `Animal`? En caso afirmativo, efectúe la modificación correspondiente.

Ejercicio 10.48 A la luz de todos los cambios que ha realizado en estas tres clases, reconsidere el grado de visibilidad de cada método y lleve a cabo los cambios que considere apropiados.

Ejercicio 10.49 ¿Ha sido posible realizar estos cambios sin afectar a ninguna otra clase del proyecto? En caso afirmativo, ¿qué es lo que esto sugiere acerca del grado de encapsulación y de acoplamiento que presentaba la versión original?

Ejercicio 10.50 *Ejercicio avanzado* Defina un tipo completamente nuevo de animal para la simulación como una subclase de `Animal`. Deberá decidir qué impacto tendrá su existencia sobre los animales existentes. Por ejemplo, el nuevo animal podría competir con los zorros como predador de la población de conejos, o ese animal podría cazar zorros pero no conejos. Probablemente compruebe que es necesario experimentar bastante con las opciones de configuración utilizadas para el nuevo animal. Tendrá que modificar el método `populate` para que se creen algunos de esos nuevos animales al principio de la simulación.

También deberá definir un nuevo color para su nueva clase de animal. Puede encontrar una lista de nombres de colores predefinidos en la página de la API que documenta la clase `Color` del paquete `java.awt`.

Ejercicio 10.51 *Ejercicio avanzado* El texto de los métodos `giveBirth` en `Fox` y `Rabbit` es muy similar. La única diferencia es que uno de ellos crea nuevos objetos `Fox` y el otro crea nuevos objetos `Rabbit`. ¿Es posible utilizar la técnica ilustrada en `canBreed` para mover el código común a un método `giveBirth` compartido en `Animal`? Si cree que es posible, compruébelo. *Sugerencia*: las reglas sobre la sustitución polimórfica se aplican tanto a los valores devueltos por los métodos, como a las asignaciones y al paso de parámetros.

10.5 Herencia múltiple

10.5.1 Una clase `Actor`

En esta sección, vamos a hablar de algunas posibles futuras ampliaciones y de ciertas estructuras de programación que permiten dar soporte a dichas ampliaciones.

La primera ampliación obvia de nuestra simulación es la adición de nuevos animales. Si ha intentado resolver el Ejercicio 10.50, entonces ya habrá tenido que enfrentar este problema. Lo que queremos aquí es generalizar un poco más: quizá no todos los participantes en la simulación serán animales. Nuestra estructura actual supone que todos los participantes que actúan en la simulación son animales y que heredan de la superclase `Animal`. Una mejora que nos gustaría poder hacer es la introducción de predadores humanos en la solución, bien como

cazadores o como tramperos. Esos actores no encajan de forma tan clara en la suposición existente de que los actores están puramente basados en algún animal. También nos gustaría ampliar la simulación para incluir las plantas de las que se alimentan los conejos, o incluso algunos aspectos del clima. Las plantas como alimento influirían en la población de conejos (en la práctica, los conejos se convierten en predadores de las plantas), y el crecimiento de las plantas podría verse influido por el clima. Estos nuevos componentes actuarían en la simulación, pero obviamente no se trata de animales, así que sería inapropiado definirlos como subclases de `Animal`.

Al considerar el potencial de introducir actores adicionales en la simulación, merece la pena revelar por qué hemos decidido almacenar los detalles de los animales tanto en un objeto `Field` como en una lista `Animal`. Visitar cada animal de la lista es lo que constituye un único paso de simulación. Colocar todos los participantes en esa única lista hace que el paso básico de simulación sea muy sencillo. Sin embargo, esto duplica claramente la información, lo que tiene el riesgo de provocar incoherencias. Una razón de esta decisión de diseño es que nos permite considerar participantes en la simulación que no se encuentren realmente dentro del hábitat –un ejemplo de ello podría ser una representación del clima.

Para tratar con actores más generales parece una buena idea introducir una superclase `Actor`. La clase `Actor` serviría como superclase de todos los tipos de participantes en la simulación, independientemente de lo que sean. En la Figura 10.3 se muestra un diagrama de clases para esta parte de la simulación. Las clases `Actor` y `Animal` son abstractas, mientras que `Rabbit`, `Fox` y `Hunter` son clases concretas.

La clase `Actor` incluiría la parte común de todos los actores. Algo que todos los posibles actores tienen en común es que realizan algún tipo de acción. También necesitamos saber si un actor sigue vivo o no. Por tanto, las únicas definiciones en la clase `Actor` son las de los métodos abstractos `act` e `isActive`:

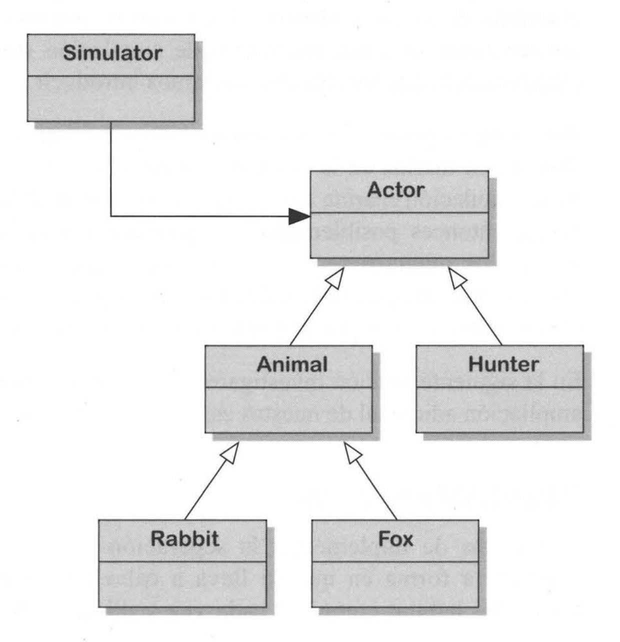


Figura 10.3
Estructura de la simulación con `Actor`.

```
// Se omiten todos los comentarios
public abstract class Actor
{
    abstract public void act(List<Actor> newActors);
    abstract public boolean isActive();
}
```

Esto debería ser suficiente para escribir de nuevo el bucle de acción en `Simulator` (Código 10.6), utilizando la clase `Actor` en lugar de la clase `Animal`. (Podríamos renombrar el método `isAlive` y denominarle `isActive` o podríamos emplear un método `isActive` separado en `Animal` que simplemente invocara al método `isAlive` existente.)

Ejercicio 10.52 Introduzca la clase `Actor` en su simulación. Escriba de nuevo el método `simulateOneStep` en `Simulator` para utilizar `Actor` en lugar de `Animal`. Puede hacer esto incluso si no ha introducido ningún nuevo tipo de participante. ¿Se compila correctamente la clase `Simulator`? ¿O hace falta algo más en la clase `Actor`?

Esta nueva estructura es más flexible porque permite añadir fácilmente actores no animales. Incluso podríamos escribir de nuevo la clase para la recopilación de estadísticas, `FieldStats`, como subclase de `Actor` –también actúa una vez en cada paso. La acción consistiría en actualizar su recuento actual de animales.

10.5.2 Flexibilidad mediante la abstracción

Al evolucionar hacia la idea de que la simulación es responsable de gestionar objetos actores, hemos conseguido abstraernos en gran medida de nuestro escenario original tan específico de los zorros y los conejos en un hábitat de forma rectangular. Este proceso de abstracción trae consigo un incremento en la flexibilidad que nos puede permitir ampliar todavía más el ámbito de lo que podríamos hacer con un entorno general de simulación. Si pensamos en los requisitos de otros escenarios de simulación similares, se nos pueden ocurrir ideas de características adicionales que queramos introducir.

Por ejemplo, podría ser útil simular otros escenarios predador-presa, como por ejemplo una simulación marina en la que estén implicados peces y tiburones, o peces y flotas pesqueras. Si la simulación marina tuviera que incluir el modelado del suministro de alimento para los peces, entonces posiblemente no querríamos visualizar las poblaciones de plancton –bien porque los números serían enormes o porque su tamaño sería demasiado pequeño. Otras simulaciones medioambientales podrían implicar modelar el clima, que a su vez, aunque es claramente un actor, puede que tampoco requiera visualización.

En la siguiente sección investigaremos la separación entre la visualización y la acción, como ampliación adicional de nuestro entorno de simulación.

10.5.3 Visualización selectiva

Una forma de implementar la separación entre la visualización y la acción consiste en cambiar la forma en que se lleva a cabo dentro de la simulación. En lugar de iterar a través del hábitat completo cada vez y dibujar a los actores en cada posición, podríamos

iterar a través de una colección separada de actores dibujables. El código en las clases del simulador tendría un aspecto similar a este:

```
// Dejar que actúen todos los actores.
for(Actor actor : actors) {
    actor.act(...);
}

// Dibujar todos los elementos dibujables.
for(Drawable item : drawables) {
    item.draw(...);
}
```

Todos los actores pertenecerían a la colección `actors`, y aquellos actores que quisiéramos mostrar en pantalla también formarían parte de la colección `drawables`. Para que esto funcione, necesitamos otra clase denominada `Drawable`, que declare un método abstracto `draw`. Los actores dibujables deben entonces heredar tanto de `Actor` como de `Drawable`. (La Figura 10.4 muestra un ejemplo en el que suponemos que tenemos hormigas (*ant*), que actúan pero que son demasiado numerosas como para visualizarlas.)

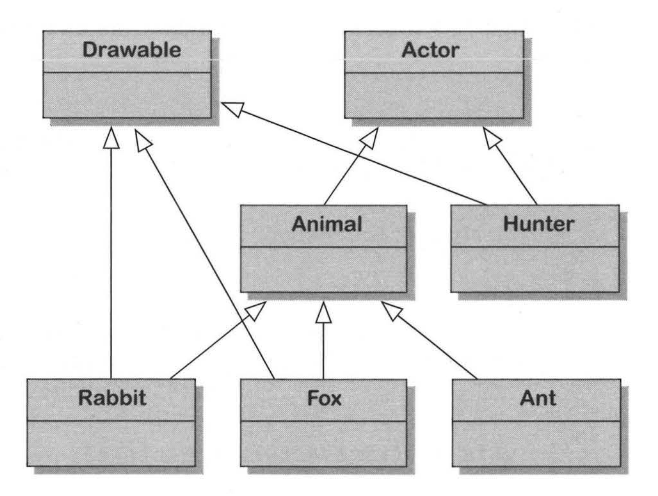


Figura 10.4
Jerarquia de actores con la clase `Drawable`.

10.5.4 Actores dibujables: herencia múltiple

> **Concepto:**
> **Herencia múltiple** Cuando tenemos una situación en la que una clase hereda de más de una superclase decimos que existe herencia múltiple.

El escenario presentado aquí utiliza una estructura denominada *herencia múltiple*. La herencia múltiple es el fenómeno que se produce en aquellos casos en los que una clase tiene más de una superclase inmediata.[4] La subclase tiene entonces todas las características de ambas subclases además de las definidas en la propia subclase.

La herencia múltiple es muy fácil de entender en principio, pero puede llevar a complicaciones significativas en la implementación de un lenguaje de programación. Los diferentes lenguajes

4 No confunda este caso con una situación normal en la que una misma clase puede tener varias superclases dentro de su jerarquía de herencia, como por ejemplo `Fox`, `Animal`, `Actor` y `Object`. No es eso lo que queremos decir al hablar del concepto de herencia múltiple.

orientados a objetos varían en su tratamiento de la herencia múltiple: algunos lenguajes permiten la herencia de múltiples superclases, mientras que otros no. Java se encuentra entre ambos casos: no permite la herencia múltiple de clases, pero proporciona otra estructura, denominada `interface`, que permite una forma limitada de herencia múltiple. Hablaremos de las interfaces en la siguiente sección.

10.6 Interfaces

Hasta este momento, hemos utilizado el término "interfaz" en un sentido informal para representar aquella parte de una clase que la acopla con las clases restantes. Java captura este concepto de una manera más formal, permitiendo definir *tipos de interfaz.*

A primera vista, las interfaces son similares a las clases, siendo la diferencia más obvia que ninguna de sus definiciones de método incluye un cuerpo de método. Por tanto, son similares a clases abstractas en las que todos los métodos sean abstractos.

10.6.1 Una interfaz `Actor`

El Código 10.9 muestra `Actor` definida como un tipo de interfaz.

Código 10.9
La interfaz `Actor`.

```
/**
 * La interfaz que deberá ser ampliada por cualquier clase
 * que quiera participar en la simulación.
 */
public interface Actor
{
    /**
     * Llevar a cabo el comportamiento normal del actor.
     * @param newActors Una lista para recibir los actores
     *                  recién creados.
     */
    void act(List<Actor> newActors);

    /**
     * ¿Está el actor todavía vivo?
     * @return true si sigue vivo, false en caso contrario.
     */
    boolean isActive();
}
```

Las interfaces Java tienen una serie de características importantes:

- Se utiliza la palabra clave `interface` en lugar de `class` en la cabecera de la declaración.
- Todos los métodos de una interfaz son abstractos; no se permite definir cuerpos para los métodos. Por tanto, la palabra clave `abstract` no es necesaria.
- Las interfaces no contienen constructores.

Concepto:

Una **interfaz** Java es una especificación de un tipo (en la forma de un nombre de tipo y un conjunto de métodos) que no define ninguna implementación para los métodos.

- Todas las cabeceras de método de una interfaz tienen visibilidad pública, por lo que no es necesaria la palabra clave `public`.
- En una interfaz solo se permite definir campos de clase constantes (`static` y `final`) con visibilidad `public`. Las palabras clave `public`, `static` y `final` pueden por tanto omitirse.

Una clase puede heredar de una interfaz de manera similar a como se hereda de una clase. Sin embargo, Java utiliza una palabra clave diferente –`implements`– para la herencia de interfaces.

Decimos que una clase *implementa* una interfaz si incluye una *cláusula implements* en su cabecera de clase. Por ejemplo:

```
public class Fox extends Animal implements Drawable
{
    // Cuerpo de la clase omitido.
}
```

Como en este caso, si una clase amplía a la vez otra clase e implementa una interfaz, entonces la cláusula `extends` debe escribirse primero en la cabecera de la clase.

Dos de nuestras clases abstractas en el ejemplo anterior, `Actor` y `Drawable`, son buenas candidatas para escribirlas como interfaces. Ambas contienen solo la definición de métodos, sin la implementación de los mismos. Por tanto, encajan ya perfectamente en la definición de una interfaz: no contienen campos de instancia, ni constructores ni cuerpos de método.

La clase `Animal` es un caso distinto. Es una clase abstracta real en el sentido de que proporciona una implementación parcial (muchos métodos tienen su correspondiente cuerpo) y solo hay un único método abstracto en su versión original. Por tanto, seguirá siendo una clase en lugar de transformarla en una interfaz.

Ejercicio 10.53 Redefina como una interfaz la clase abstracta `Actor` en su proyecto. ¿Se sigue compilando la simulación? ¿Se ejecuta? Efectúe cualquier cambio necesario para que vuelva a ser ejecutable.

Ejercicio 10.54 Los campos de la siguiente interfaz, ¿son campos de clase o campos de instancia? ¿Qué visibilidad tienen?

```
public interface Quiz
{
    int CORRECT = 1;
    int INCORRECT = 0;
    ...
}
```

Ejercicio 10.55 ¿Qué errores hay en la siguiente interfaz?

```
public interface Monitor
{
    private static final int THRESHOLD = 50;
    public Monitor (int initial);
    public int getThreshold()
    {
        return THRESHOLD;
    }
    ...
}
```

10.6.2 Herencia múltiple de interfaces

Como hemos mencionado anteriormente, Java permite a una clase ampliar como mucho otra clase. Sin embargo, sí que permite que una clase implemente cualquier número de interfaces (además de ampliar, posiblemente, otra clase). Por tanto, si definimos tanto `Actor` como `Drawable` como interfaces en lugar de como clases abstractas, podremos definir la clase `Hunter` (Figura 10.4) para que implemente las dos:

```
public class Hunter implements Actor, Drawable
{
    // Cuerpo de la clase omitido.
}
```

La clase `Hunter` hereda los métodos de todas las interfaces (`act` y `draw`, en este caso) como métodos abstractos. Por tanto, debe proporcionar definiciones de método para los dos, sustituyendo los métodos indicados, porque de lo contrario habrá que declarar como abstracta la propia clase.

La clase `Animal` muestra un ejemplo en el que una clase no implementa un método de una interfaz heredada. `Animal`, en nuestra nueva estructura de la Figura 10.4, hereda el método abstracto `act` de `Actor`. No proporciona ningún cuerpo para este método, lo que hace que la propia `Animal` sea una clase abstracta (deberá incluir la palabra clave `abstract` en la cabecera de la clase).

Por su parte, las subclases de `Animal` sí que implementan el método `act` con lo que se convierten en clases concretas.

Ejercicio 10.56 *Ejercicio avanzado* Añada un actor no animal a la simulación. Por ejemplo, podría introducir una clase `Hunter` que represente a los cazadores y que presente las siguientes propiedades: los cazadores no tienen edad máxima y no se alimentan ni se reproducen. En cada paso de la simulación, un cazador se mueve a una posición aleatoria situada en cualquier lugar del hábitat y hace un número fijo de disparos hacia posiciones aleatorias situadas dentro del hábitat. Cualquier animal que se encuentre en una de esas posiciones morirá.

Coloque solo un pequeño número de cazadores en el hábitat al principio de la simulación. ¿Permanecen los cazadores durante toda la simulación o alguna vez llegan a desaparecer? Si desaparecen, ¿por qué podría ser? ¿Cree que eso representa un comportamiento realista?

¿Qué otras clases han necesitado una modificación como resultado de la introducción de los cazadores? ¿Es necesario introducir un mayor nivel de desacoplamiento en esas clases?

10.6.3 Interfaces como tipos

Cuando una clase implementa una interfaz, no hereda ninguna implementación de ella, porque las interfaces no pueden contener cuerpos de método. La cuestión es entonces: ¿qué es lo que ganamos en la práctica implementando interfaces?

Cuando presentamos la herencia en el Capítulo 8, destacamos dos grandes ventajas de la misma:

- La subclase hereda el código (implementaciones de métodos y campos) de la superclase. Esto permite reutilizar el código existente y evita la duplicación de código,

- La subclase se convierte en un subtipo de la superclase. Esto permite disponer de variables y llamadas de métodos polimórficas. En otras palabras, permite que diferentes casos especiales de objetos (instancias de subclases) sean tratados de manera uniforme (como instancias del supertipo).

Las interfaces no proporcionan la primera ventaja (porque no contienen implementaciones), pero sí que proporcionan la segunda. Una interfaz define un tipo exactamente igual que lo hace una clase. Esto significa que pueden declararse variables para que sean del tipo de la interfaz, aun cuando no puedan existir objetos de dicho tipo (solo de los subtipos).

En nuestro ejemplo, aunque `Actor` es ahora una interfaz, podemos seguir declarando una variable `Actor` en la clase `Simulator`. El bucle de simulación seguirá funcionando sin necesidad de que efectuemos ninguna modificación.

Las interfaces no pueden tener instancias directas, pero sirven como supertipos para las instancias de otras clases.

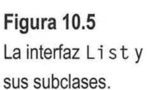
10.6.4 Interfaces como especificaciones

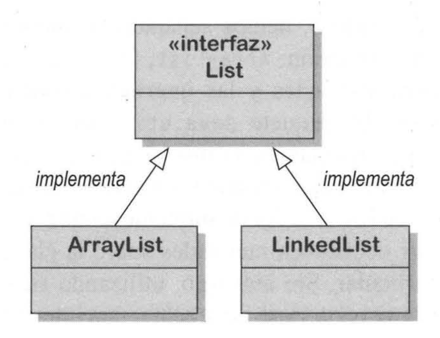
En este capítulo hemos presentado las interfaces como medio de implementar la herencia múltiple en Java. Este es uno de los usos importantes de las interfaces, pero tienen también otros usos.

La característica más importante de las interfaces es que separan completamente la definición de la funcionalidad (la "interfaz" de la clase en el sentido más amplio de la palabra) con respecto a su implementación. Podemos encontrar en la jerarquía de colecciones de Java un buen ejemplo de cómo puede emplearse esto en la práctica.

La jerarquía de colecciones define (entre otros tipos) la interfaz `List` y las clases `ArrayList` y `LinkedList` (Figura 10.5). La interfaz `List` especifica la funcionalidad completa de una lista, sin proporcionar ninguna implementación. Las subclases (`LinkedList` y `ArrayList`) proporcionan dos implementaciones distintas de la misma interfaz. Esto es interesante, porque las dos implementaciones difieren enormemente en la eficiencia de algunas de sus funciones. El acceso aleatorio a los elementos situados en la mitad de la lista, por ejemplo, es mucho más rápido con `ArrayList`. Sin embargo, en `LinkedList`, la inserción o el borrado de elementos pueden ser mucho rápidos.

Qué implementación será mejor en cualquier aplicación dada es algo que puede ser difícil de juzgar de antemano. Depende en gran medida de la frecuencia relativa con la que se efectúen ciertas operaciones, así como de algunos otros factores. En la práctica, la mejor forma de


Figura 10.5
La interfaz `List` y sus subclases.



averiguarlo suele ser probar las distintas soluciones: implementar la aplicación con ambas alternativas y medir el rendimiento.

La existencia de la interfaz `List` hace que sea muy sencillo llevar esto a cabo. Si en lugar de utilizar `ArrayList` o `LinkedList` como tipos para las variables y los parámetros, utilizamos siempre `List`, nuestra aplicación funcionará independientemente del tipo específico de lista que estemos utilizando actualmente. Solo tenemos que emplear realmente el nombre de la implementación específica cuando creemos una nueva lista. Por ejemplo, lo que haríamos es escribir:

```
List<Type> myList = new ArrayList<Type>();
```

Observe que el tipo de la variable polimórfica es simplemente `List` de `Type`. De esta forma, podemos cambiar la aplicación completa para que use una lista enlazada, simplemente cambiando `ArrayList` por `LinkedList` en una única ubicación: aquella en la que se está creando la lista.

Ejercicio 10.57 ¿Qué métodos tienen `ArrayList` y `LinkedList` que no estén definidos en la interfaz `List`? ¿Por qué cree que estos métodos no están incluidos en `List`?

Ejercicio 10.58 Escriba una clase que realice comparaciones entre la eficiencia de los métodos comunes de la interfaz `List` existentes en las clases `ArrayList` y `LinkedList`, como `add`, `get` y `remove`. Aplique la técnica de la variable polimórfica que hemos descrito anteriormente para escribir la clase, de modo que esta solo sepa que está llevando a cabo sus pruebas con objetos del tipo de interfaz `List`, en lugar de trabajar con los tipos concretos `ArrayList` y `LinkedList`. Utilice listas de objetos de gran tamaño en las pruebas para que los resultados sean significativos. Puede emplear el método `currentTimeMillis` de la clase `System` para averiguar los instantes inicial y final de la ejecución de sus métodos de prueba.

Ejercicio 10.59 Lea la descripción de la API correspondiente a los métodos `sort` de la clase `Collections` en el paquete `java.util`. ¿Qué interfaces se mencionan en las descripciones?

Ejercicio 10.60 *Ejercicio avanzado* Investigue la interfaz `Comparable`. Se trata de una interfaz *parametrizada*. Defina una clase que implemente `Comparable`. Cree una colección que contenga objetos de esta clase y ordene la colección. *Sugerencia*: la clase `LogEntry` del proyecto *weblog-analyzer* del Capítulo 4 implementa esta interfaz.

10.6.5 Soporte de librería mediante clases abstractas e interfaces

En el Capítulo 5, hemos señalado la importancia de prestar atención a los nombres de las clases de colección: `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, etc. Ahora que hemos visto las clases abstractas y las interfaces, podemos ver por qué se han elegido estos nombres concretos. El paquete `java.util` define en forma de interfaces varias abstracciones de colección importantes, como `List`, `Map` y `Set`. Los nombres de las clases concretas se han elegido para proporcionar información sobre a qué tipo de interfaz se ajustan y acerca de algunos de los detalles de implementación subyacentes. Esta información es muy útil a la hora de tomar decisiones racionales sobre la clase concreta correcta que hay que utilizar en cada caso particular. Sin embargo, utilizando siempre que sea posible el tipo abstracto de mayor nivel (ya sea una clase abstracta o una interfaz) para nuestras variables, nuestro código seguirá

siendo flexible de cara a futuros cambios en la librería –como por ejemplo la adición de una nueva implementación de `Map` o `Set`.

En el Capítulo 11, donde veremos las librerías GUI de Java, haremos un gran uso de las clases abstractas y de las interfaces, a medida que veamos cómo crear funcionalidad enormemente sofisticada con muy poco código adicional.

10.7 Otro ejemplo más de interfaces

En la sección anterior hemos explicado cómo pueden utilizarse las interfaces para separar la especificación de un componente de su implementación, de forma que se puedan "conectar" diferentes implementaciones, facilitando así la sustitución de componentes dentro de un sistema. Esto suele hacerse con partes separadas de un sistema que estén solo débilmente acopladas desde un punto de vista lógico.

Hemos visto un ejemplo de esto (sin analizarlo de manera explícita) al final de la Sección 10.3. Allí, investigamos el proyecto *foxes-and-rabbits-graph*, que añadía otra vista de las poblaciones en forma de gráfica lineal. Examinando el diagrama de clases de ese proyecto, podemos ver que la adición hace también uso de una interfaz Java (Figura 10.6).

Las versiones previas del proyecto *foxes-and-rabbits* contenían solo una clase `SimulatorView`. Era una clase concreta, que proporcionaba la implementación de una vista del hábitat en forma de cuadrícula. Como hemos visto, la visualización está bastante separada de la lógica de simulación (el hábitat y los actores) y resulta posible disponer de diferentes vistas para presentar los resultados.

Para este proyecto, `SimulatorView` se modificó pasando de ser una clase a una interfaz, y la implementación de esa vista se pasó a una clase denominada `GridView`.

`GridView` es idéntica a la clase `SimulatorView` anterior. La nueva interfaz `SimulatorView` se construyó buscando en la clase `Simulator` todos los métodos que son invocados en la práctica desde fuera y luego definiendo una interfaz que especifica exactamente dichos métodos. Se trata de lo siguiente:

```
view.setColor(classObject, color);
view.isViable(field);
view.showStatus(step, field);
view.reset();
```

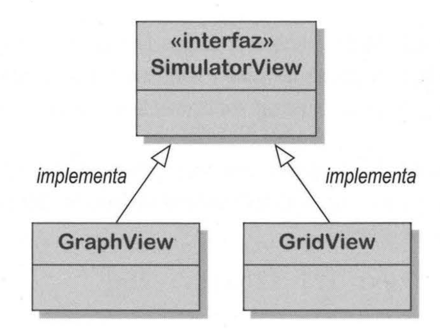


Figura 10.6
La interfaz `SimulatorView` y las clases que la implementan.

Ahora podemos definir fácilmente la interfaz `SimulatorView` completa:

```
import java.awt.Color;
public interface SimulatorView
{
    void setColor(Class animalClass, Color color);
    boolean isViable(Field field);
    void showStatus(int step, Field field);
    void reset();
}
```

El único detalle ligeramente complejo de la definición anterior es el uso del tipo `Class` como el primer parámetro del método `setColor`. Explicaremos esto en la siguiente sección.

La clase `SimulatorView` anterior, ahora denominada `GridView`, se especifica de manera que implemente la nueva interfaz `SimulatorView`:

```
public class GridView extends JFrame implements SimulatorView
{
    ...
}
```

No requiere ningún código adicional, porque ya implementa los métodos de la interfaz. Sin embargo, después de realizar estos cambios es bastante sencillo "conectar" otras vistas de la simulación, proporcionando implementaciones adicionales de la interfaz `SimulatorView`. La nueva clase `GraphView`, que genera la gráfica de líneas, es un ejemplo de esto.

Una vez que disponemos de más de una implementación de la vista, podemos sustituir fácilmente la vista actual por otra o, como vemos en el ejemplo, mostrar incluso dos vistas al mismo tiempo. En la clase `Simulator`, las subclases concretas `GridView` y `GraphView` solo se mencionan una vez, en el momento de construir cada vista. Después, se guardan en una colección que almacena elementos del supertipo `SimulatorView`, utilizándose únicamente el tipo de interfaz para comunicarse con ellas.

Las implementaciones de las clases `GridView` y `GraphView` son bastante complejas, y no esperamos que sea capaz de comprenderlas completamente por el momento. Sin embargo, el patrón consistente en proporcionar dos implementaciones para una única interfaz sí que es importante aquí, así que asegúrese de que comprende perfectamente este aspecto.

Ejercicio 10.61 Repase el código fuente de la clase `Simulator`. Localice todas las apariciones de las interfaces y clases de vista y examine todas las variables declaradas utilizando cualquiera de estos tipos. Explique cómo se utilizan exactamente las vistas en la clase `Simulator`.

Ejercicio 10.62 Diseñe una nueva clase `TextView` que implemente `SimulatorView`. `TextView` debe proporcionar una vista textual de la simulación. Después de cada paso de simulación, tiene que imprimir una línea de la forma

```
Foxes: 121 Rabbits: 266
```

Utilice `TextView` en lugar de `GridView` para algunas pruebas. (No borre la clase `GridView`. ¡Queremos tener la posibilidad de cambiar entre diferentes vistas!)

Ejercicio 10.63 ¿Puede conseguir disponer de las tres vistas activas al mismo tiempo?

10.8 La clase `Class`

En el Capítulo 8, hemos descrito la clase `Object`, que tiene un nombre bastante paradójico. ¡No debería sorprenderle, por tanto, que también exista una clase denominada `Class`! Llegados a este punto, es cuando el hablar acerca de clases y de objetos puede resultar algo confuso.

Hemos utilizado el tipo `Class` en la definición de la interfaz `SimulatorView` en la sección anterior. La clase `Class` no tiene nada que ver específicamente con las interfaces; se trata de una característica general de Java, pero lo que pasa es que nos acabamos de topar con ella por primera vez al hablar de las interfaces. La idea es que cada tipo tiene un objeto `Class` asociado con él.

La clase `Object` define el método `getClass` para devolver la instancia de `Class` asociada con un objeto. Otra forma de obtener el objeto `Class` para un cierto tipo consiste en escribir "`.class`" detrás del nombre del tipo: por ejemplo, `Fox.class` o `int.class` –observe que incluso los tipos primitivos tienen objetos `Class` asociados.

Los objetos `Class` son particularmente útiles si deseamos saber si el tipo de dos objetos coincide. Utilizamos esta característica en la clase original `SimulatorView` para asociar cada tipo de animal con un color dentro del hábitat. `SimulatorView` dispone del siguiente campo para establecer dicha asociación:

```
private Map<Class, Color> colors;
```

En el momento de configurar la vista, el constructor de `Simulator` realiza las siguientes llamadas a su método `setColor`:

```
view.setColor(Rabbit.class, Color.ORANGE);
view.setColor(Fox.class, Color.BLUE);
```

No vamos a entrar en más detalles sobre `Class`, pero esta descripción debería ser suficiente para permitirle comprender el código mostrado en la sección anterior.

10.9 ¿Clase abstracta o interfaz?

En algunas situaciones, hay que tomar una decisión sobre si emplear una clase abstracta o una interfaz. A veces, la decisión es sencilla: cuando se pretende que la clase contenga implementaciones de algunos métodos, necesitaremos utilizar una clase abstracta. En otros casos, pueden emplearse clases abstractas o interfaces.

Si podemos elegir, las interfaces suelen ser preferibles. Si proporcionamos un tipo en forma de clase abstracta, entonces las subclases no pueden ampliar ninguna otra clase. Puesto que las

interfaces permiten la herencia múltiple, el uso de una interfaz no impone dicha restricción. Por tanto, la utilización de interfaces permite obtener una estructura más flexible y más ampliable.

10.10 Simulaciones dirigidas por sucesos

El estilo de simulación que hemos utilizado en este capítulo tiene la característica de que el paso del tiempo se produce en pasos discretos de igual longitud. En cada paso temporal, se le pide a cada actor de la simulación que actúe –es decir, que lleve a cabo las acciones apropiadas según su estado actual. Este estilo de simulación en ocasiones se denomina simulación *de carácter temporal* o *síncrona*. En esta simulación en particular, la mayoría de los actores tenían algo que hacer en cada paso temporal: moverse, reproducirse y alimentarse. Sin embargo, en muchos otros escenarios de simulación, los actores invierten un gran número de pasos temporales no haciendo nada –normalmente, esperan a que suceda algo que requiera una cierta acción por su parte. Considere por ejemplo el caso de un conejo recién nacido en nuestra simulación. A ese conejo se le pregunta repetidamente si va a reproducirse, aunque se necesitan muchos pasos temporales antes de que esto sea posible. ¿Existe alguna manera de evitar plantear esta cuestión innecesaria, hasta que el conejo esté realmente listo?

Por otro lado, tenemos la cuestión de cuál es el tamaño más apropiado del paso temporal. Hemos elegido ser deliberadamente vagos en lo que respecta a cuánto tiempo real representa cada paso temporal y las distintas acciones requieren en realidad cantidades significativamente distintas de tiempo (comer y moverse es algo que tendría que ocurrir mucho más frecuentemente que, por ejemplo, reproducirse). ¿Hay alguna forma de elegir un tamaño de paso temporal que no sea demasiado pequeño como para que la mayor parte del tiempo no suceda nada, ni demasiado largo como para que los diferentes tipos de acciones no se distingan de forma suficientemente clara entre un paso temporal y otro?

Un enfoque alternativo consiste en emplear un estilo de simulación *dirigida por sucesos* o *asíncrona*. En este estilo, la simulación se dirige manteniendo una planificación de sucesos futuros. La diferencia más obvia entre ambos estilos es que en una simulación dirigida por sucesos, el tiempo transcurre en cantidades variables. Por ejemplo, un suceso puede ocurrir en el instante t y los siguientes dos sucesos ocurrir en los instantes $t+2$ y $t+8$, mientras que los tres sucesos siguientes podrían tener lugar todos ellos en el instante $t+9$.

Para una simulación de zorros y conejos, el tipo de sucesos de los que estamos hablando serían el nacimiento, el movimiento, la caza y la muerte debida a causas naturales. Lo que ocurre normalmente, es que a medida que se produce cada suceso se planifica un nuevo suceso para algún momento futuro. Por ejemplo, cuando se produce un nacimiento, se planifica el suceso que marca la muerte del animal debido a que alcanza su edad máxima. Todos los sucesos futuros se almacenan en una cola ordenada, en la que el siguiente suceso que tiene que producirse se mantiene en la cabeza de la cola. Es importante observar que los sucesos recién planificados no siempre se colocarán al final de la cola actual; a menudo tendrán que insertarse en algún lugar en mitad de la cola, para mantener esta en orden temporal. Además, algunos sucesos futuros quedarán obsoletos debido a sucesos que se producen antes de ellos –un ejemplo obvio sería que el suceso de muerte natural de un conejo no tendrá lugar si es comido con anterioridad.

Las simulaciones dirigidas por sucesos se prestan particularmente bien a las técnicas que hemos descrito en este capítulo. Por ejemplo, el concepto de suceso es probable que se implemente como una clase abstracta `Event` que contenga detalles concretos acerca de cuándo tendrá lugar el suceso, pero que solo contendrá detalles abstractos sobre lo que implica ese suceso. Las subclases concretas de `Event` suministrarán entonces los detalles específicos para los distintos tipos de sucesos. Normalmente, el bucle principal de simulación no tendrá que preocuparse por los tipos concretos de sucesos, sino que será capaz de usar llamadas a métodos polimórficos cada vez que se produzca un suceso.

Las simulaciones dirigidas por sucesos suelen ser más eficientes y resultan preferibles cuando tenemos que tratar con sistemas de gran tamaño y con grandes cantidades de datos, mientras que las simulaciones síncronas resultan mejores para generar visualizaciones de carácter temporal (como por ejemplo animaciones de los actores), porque el tiempo fluye de manera más uniforme.

Ejercicio 10.64 Obtenga más información sobre las diferencias entre las simulaciones dirigidas por sucesos y las simulaciones de carácter temporal.

Ejercicio 10.65 Examine el paquete `java.util` para ver si hay alguna clase que pudiera resultar adecuada para almacenar una cola de sucesos en una simulación dirigida por sucesos.

Ejercicio 10.66 *Ejercicio avanzado* Escriba de nuevo la simulación de los zorros y los conejos en el estilo basado en sucesos.

10.11 Resumen sobre la herencia

En los tres capítulos anteriores hemos expuesto muchos aspectos distintos de las técnicas de herencia. Entre ellos se incluyen la herencia de código y los subtipos, así como la herencia de interfaces, las clases abstractas y la clases concretas.

En general, podemos distinguir dos objetivos principales de la utilización de la herencia: podemos emplearla para heredar código (herencia de código) y para heredar el tipo (subtipos). El primer objetivo es útil para reutilizar el código, mientras que el segundo lo es de cara al polimorfismo y la especialización.

Cuando heredamos de ("extendemos" o "ampliamos") clases concretas, hacemos las dos cosas: heredar la implementación y el tipo. Cuando heredamos de ("implementamos") interfaces, separamos las dos cosas: heredamos un tipo, pero ninguna implementación. Para casos en los que haya partes de ambos objetivos que resulten útiles, podemos heredar de clases abstractas; en ese caso, heredamos el tipo y una implementación parcial.

Cuando heredamos una implementación completa, podemos decidir optar por añadir o sustituir métodos. Cuando no se hereda ninguna implementación de un tipo, o solo se hereda una implementación parcial, la subclase debe proporcionar la implementación antes de poder instanciarla.

Algunos otros lenguajes orientados a objetos también proporcionan mecanismos para heredar código sin heredar el tipo. Java no proporciona tal estructura.

10.12 Resumen

En este capítulo hemos explicado la estructura fundamental de las simulaciones por computadora. A continuación, hemos utilizado un ejemplo para presentar las clases abstractas y las interfaces, como estructuras que nos permiten crear abstracciones adicionales y desarrollar aplicaciones más flexibles.

Las clases abstractas son clases que no están pensadas para disponer de ninguna instancia. Su propósito es servir como superclases de otras clases. Las clases abstractas pueden tener tanto métodos abstractos (métodos que tiene cabecera pero no cuerpo) como implementaciones completas de métodos. Las subclases concretas de las clases abstractas deben sustituir los métodos abstractos, para proporcionar las implementaciones de estos métodos abstractos que faltan.

Otra estructura para definir tipos en Java es la interfaz. Las interfaces en Java son similares a clases completamente abstractas: definen cabeceras de métodos, pero sin proporcionar ninguna implementación. Las interfaces definen tipos que se pueden usar para las variables.

Las interfaces se pueden emplear para proporcionar la especificación de una clase (o de parte de una aplicación) sin decir nada acerca de la implementación concreta.

Java permite la herencia múltiple de interfaces (lo que denomina relaciones *implements*) pero en el caso de clases solo permite la herencia simple (relaciones *extends*).

Términos introducidos en el capítulo

método abstracto, clase abstracta, clase concreta, subclase abstracta, herencia múltiple, interfaz (estructura Java), implements

Resumen de conceptos

- **método abstracto** La definición de un método abstracto está compuesta de una cabecera de método, sin que existe cuerpo del mismo. Se marca con la palabra clave `abstract`.
- **clase abstracta** Una clase abstracta es una clase que no está pensada para crear instancias. Su objetivo es servir como una superclase de otras clases. Las clases abstractas pueden contener métodos abstractos.
- **subclase abstracta** Para que una subclase de una clase abstracta se transforme en concreta, debe proporcionar implementaciones para todos los métodos abstractos heredados. En caso contrario, la propia subclase será también abstracta.
- **llamadas a métodos en la superclase** Las llamadas a métodos de instancia no privados desde dentro de una superclase siempre se evalúan en el contexto más amplio del tipo dinámico del objeto.
- **herencia múltiple** Cuando tenemos una situación en la que una clase hereda de más de una superclase decimos que existe herencia múltiple.
- **interfaz** Una interfaz Java es una especificación de un tipo (en la forma de un nombre de tipo y un conjunto de métodos) que no define ninguna implementación para los métodos.

Ejercicio 10.67 ¿Puede tener una clase abstracta métodos concretos (no abstractos)? ¿Puede una clase concreta tener métodos abstractos? ¿Podemos tener una clase abstracta sin métodos abstractos? Justifique sus respuestas.

Ejercicio 10.68 Examine el siguiente código. Tenemos cinco tipos –clases o interfaces– (U, G, B, Z y X) y una variable de cada uno de esos tipos.

```
U u;
G g;
B b;
Z z;
X x;
```

Las siguientes asignaciones son todas legales (suponemos que todas se compilan).

```
u = z;
x = b;
g = u;
x = u;
```

Las siguientes asignaciones son todas ilegales (producen errores de compilación).

```
u = b;
x = g;
b = u;
z = u;
g = x;
```

¿Qué podemos deducir acerca de los tipos y sus relaciones? (¿Qué relación existe entre ellas?)

Ejercicio 10.69 Suponga que queremos modelar las personas de una universidad para implementar un sistema de gestión de cursos. Hay distintas personas implicadas: miembros del personal, estudiantes, personal docente, personal de soporte, tutores, personal de soporte técnico y técnicos estudiantes. Los tutores y los técnicos estudiantes son interesantes: los tutores son estudiantes que han sido contratados para realizar una cierta labor docente, mientras que los técnicos estudiantes son estudiantes que han sido contratados para ayudar en las tareas de soporte técnico.

Dibuje una jerarquía de tipos (clases e interfaces) para representar esta situación. Indique qué tipos son clases concretas, clases abstractas e interfaces.

Ejercicio 10.70 *Ejercicio avanzado* En ocasiones, existen parejas clase/interfaz en la librería estándar Java que definen exactamente los mismos métodos. A menudo, el nombre de la interfaz termina con *Listener* y el nombre de la clase con *Adapter.* Un ejemplo sería `PrintJobListener` y `PrintJobAdapter`. La interfaz define algunas cabeceras de métodos y la clase adaptadora define los mismos métodos, pero cada uno con un cuerpo de método vacío. ¿Cuál podría ser la razón para tener ambos?

Ejercicio 10.71 La librería de colecciones tiene una clase denominada `TreeSet`, que es un ejemplo de un conjunto ordenado. Los elementos de este conjunto se mantienen en orden. Lea cuidadosamente la descripción de esta clase y luego escriba una clase `Person` que pueda insertarse en un `TreeSet`, que luego ordenará los objetos `Person` según la edad.

Ejercicio 10.72 Utilice la documentación de la API para la clase `AbstractList` con el fin de escribir una clase concreta que mantenga una lista no modificable.

CAPÍTULO

11 Construcción de interfaces gráficas de usuario

Principales conceptos explicados en el capítulo:

- construcción de interfaces GUI
- componentes de una interfaz
- diseño gráfico de la GUI
- tratamiento de sucesos

Estructuras Java explicadas en este capítulo:

`JFrame`, `JLabel`, `JButton`, `JMenuBar`, `JMenu`, `JMenuItem`, `ActionEvent`, `Color`, `FlowLayout`, `BorderLayout`, `GridLayout`, `BoxLayout`, `Box`, `JOptionPane`, `EtchedBorder`, `EmptyBorder`, clases internas anónimas, variables `final`

11.1 Introducción

Hasta ahora, en el libro nos hemos concentrado en escribir aplicaciones con interfaces basadas en texto. La razón no es que estas interfaces ofrezcan, en principio, ninguna gran ventaja, la única ventaja que tienen es que son más fáciles de crear.

Además, no queríamos distraer demasiado la atención de las cuestiones más importantes del desarrollo software, en estas primeras etapas de aprendizaje de la programación orientación a objetos. Hemos preferido concentrarnos en problemas como la interacción y estructura de los objetos, el diseño de clases y la calidad del código.

Las interfaces gráficas de usuario (a partir de ahora GUI, del inglés *Graphical User Interface*) también se construyen a partir de objetos que interactúan, pero tienen una estructura muy especializada, y hemos preferido evitar presentarlas antes de analizar las estructuras de los objetos en términos más generales. Sin embargo, ahora sí que estamos listos para echar un vistazo a las técnicas de construcción de interfaces GUI.

Las interfaces GUI proporcionan a nuestras aplicaciones una interfaz compuesta de ventanas, menús, botones y otros componentes gráficos. Hacen que las aplicaciones se parezcan mucho más a la "típica" aplicación que la mayoría de las personas utilizan hoy día.

Observe que nos topamos de nuevo aquí con el doble significado de la palabra *interfaz*. Las interfaces de las que estamos hablando ahora no son ni las interfaces de las clases ni la

estructura `interface` de Java. Ahora estamos hablando de *interfaces de usuario* –la parte de una aplicación que es visible en pantalla para que el usuario pueda interaccionar con ella.

Una vez que sepamos cómo crear interfaces GUI con Java, podremos desarrollar programas con un aspecto mucho mejor.

11.2 Componentes, diseño y tratamiento de sucesos

Los detalles implicados en la creación de interfaces GUI son muy prolijos. En este libro no nos será posible cubrir todos los detalles de todas las posibles cosas que pueden hacerse con esas interfaces, pero sí que expondremos los principios generales e incluiremos un buen número de ejemplos.

Toda la programación de interfaz GUI en Java se realiza utilizando librerías estándar de clases dedicadas. Una vez que comprendamos los principios, nos resultará fácil encontrar todos los detalles necesarios, trabajando con la documentación de la librería estándar.

Los principios que necesitamos comprender pueden dividirse en tres áreas temáticas:

- ¿Qué tipo de elementos podemos mostrar en pantalla?
- ¿Cómo colocamos esos elementos?
- ¿Cómo podemos reaccionar a la entrada del usuario?

Estas cuestiones se corresponden con tres conceptos clave: *componentes, diseño gráfico* y *tratamiento de sucesos.*

Concepto:

Una interfaz GUI se construye disponiendo **componentes** en pantalla. Los componentes se representan mediante objetos.

Los *componentes* son las partes individuales con las que se construye una GUI. Son cosas como botones, menús, elementos de menú, casillas de verificación, barras de desplazamiento, campos de texto, etc. La librería Java contiene un buen número de componentes prefabricados, y además podemos escribir nuestros propios componentes. Tendremos que aprender cuáles son los componentes más importantes, cómo crearlos y cómo hacer que tengan el aspecto que deseamos.

Concepto:

Para definir la colocación de los componentes de una GUI se utilizan gestores de diseño gráfico.

El *diseño gráfico* se ocupa de la cuestión de cómo disponer los componentes en pantalla. Los sistemas GUI más antiguos y primitivos solucionaban este tema mediante coordenadas bidimensionales: el programador especificaba coordenadas *x* e *y* (en píxeles) para indicar la posición y el tamaño de cada componente. En los sistemas GUI más mordernos, sin embargo, esto es demasiado simplista. Tenemos que tener en cuenta las diferentes resoluciones de pantalla, los distintos tipos de fuente, el hecho de que los usuarios pueden cambiar el tamaño de las ventanas y muchos otros aspectos que hacen que el diseño gráfico de la GUI resulte más difícil. La solución será un esquema en el que podemos especificar ese diseño en términos más generales. Por ejemplo, podemos especificar que un determinado componente debe estar debajo de otro o que el tamaño de un cierto componente debe ser reducido si se redimensiona la ventana, mientas que tal otro componente debe tener siempre un tamaño fijo. Más adelante veremos que esta tarea se lleva a cabo utilizando *gestores de diseño gráfico.*

Concepto:

El término **tratamiento de sucesos** hace referencia a la tarea de reaccionar a los sucesos provocados por el usuario, como los clics de ratón o la entrada a través del teclado.

El *tratamiento de sucesos* hace referencia a la técnica que utilizaremos para tratar con la entrada del usuario. Una vez que hayamos creado nuestros componentes y los hayamos colocado en pantalla, también tendremos que asegurarnos de que ocurra algo cuando el usuario haga clic en un botón. El modelo utilizado por la librería Java para resolver este tema

está basado en sucesos: si un usuario activa un componente (por ejemplo, hace clic en un botón o selecciona un elemento de menú) el sistema generará un suceso. Nuestra aplicación puede entonces recibir una notificación del suceso (haciendo que sea invocado uno de sus métodos), con lo que podremos llevar a cabo la acción apropiada.

Posteriormente en el capítulo hablaremos con mucho más detalle acerca de cada una de estas áreas. Sin embargo, en primer lugar vamos a presentar brevemente algo de información de referencia adicional, junto con la necesaria terminología.

11.3 AWT y Swing

Java tiene dos librerías GUI. La más antigua se denomina AWT (*Abstract Window Toolkit*) y fue introducida como parte de la API Java original. Más tarde se añadió a Java una librería GUI muy mejorada, denominada *Swing*.

Swing hace uso de algunas de las clases de AWT, sustituye algunas de las clases de AWT por su propia versión y añade muchas nuevas clases (Figura 11.1).

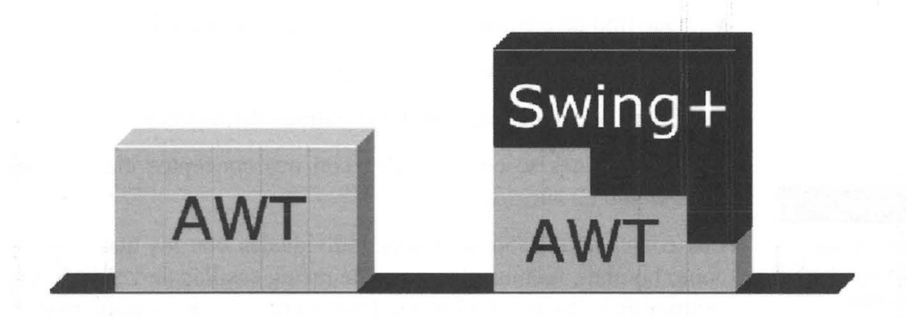


Figura 11.1
AWT y Swing.

En este libro, vamos a emplear las librerías Swing. Esto significa que utilizaremos algunas clases de AWT que todavía se usan con programas Swing, aunque usaremos las versiones Swing de todas las clases que existan en ambas librerías.

Cuando existen clases equivalentes en AWT y Swing, las versiones Swing se han identificado añadiendo la letra mayúscula *J* al principio del nombre de la clase. Por ejemplo, en la documentación podrá encontrar clases denominadas `Button` y `JButton`, `Frame` y `JFrame`, `Menu` y `JMenu`, etc. Las clases que comienzan con *J* son las versiones Swing; esas son las que usaremos y no deben mezclarse las dos versiones en una misma aplicación.

Con esto tenemos ya la suficiente información de referencia para poder comenzar. Examinemos algo de código.

11.4 El ejemplo ImageViewer

Como siempre, explicaremos los conceptos nuevos analizando un ejemplo. La aplicación que construiremos en este capítulo es un visualizador de imágenes (Figura 11.2). Es un programa que puede abrir y mostrar archivos de imagen en formatos JPEG y PNG, realizar algunas transformaciones en las imágenes y guardarlas en disco.

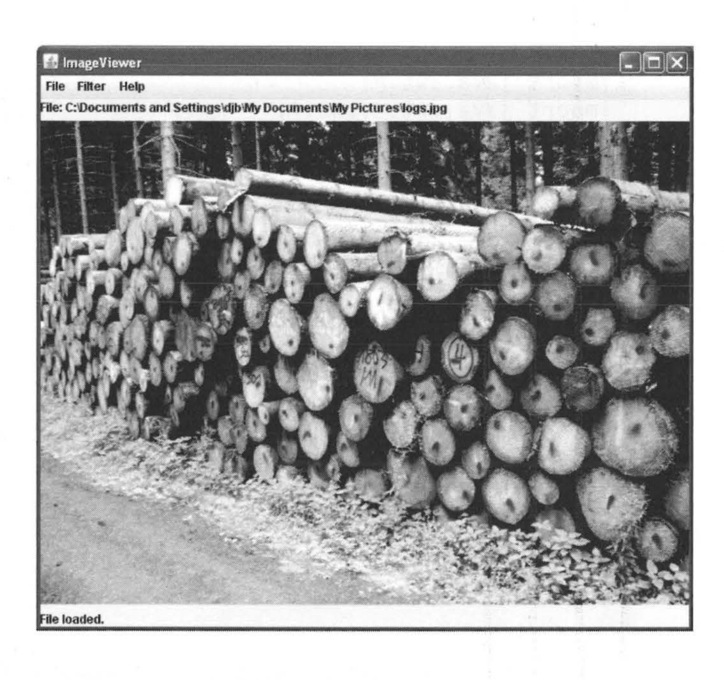


Figura 11.2
Una aplicación simple de visualización de imágenes.

Como parte de esta tarea, utilizaremos nuestra propia clase para representar una imagen mientras se encuentra en memoria, implementaremos varios filtros para modificar la apariencia de la imagen y emplearemos componentes Swing para construir una interfaz de usuario. Mientras hacemos esto, centraremos las explicaciones en los aspectos del programa relativos a la GUI.

> **Concepto:**
>
> **Formato de imagen** Las imágenes pueden almacenarse en diferentes formatos. Las diferencias afectan principalmente al tamaño del archivo y a la calidad de la imagen.

Si tiene curiosidad por lo que vamos a construir, puede abrir y probar el proyecto *imageviewer1-0*, que es la versión mostrada en la Figura 11.2; pruebe a crear un objeto `ImageViewer`. Hay disponibles algunas imágenes de ejemplo en la subcarpeta *images* de la carpeta *chapter11* (situada un nivel más arriba de la carpeta del proyecto). Por supuesto, también puede abrir sus propias imágenes. Aquí, vamos a comenzar poco a poco, empezando con algo mucho más simple y progresando paso a paso hacia la aplicación final.

11.4.1 Primeros experimentos: creación de un marco

Casi todo lo que podemos ver en una GUI está contenido en una ventana de nivel superior. Una ventana de nivel superior es aquella que se encuentra bajo control del gestor de ventanas del sistema operativo y que normalmente puede moverse, redimensionarse, minimizarse y maximizarse de forma independiente.

Java denomina a estas ventanas de nivel superior *marcos*. En Swing, se representan mediante una clase llamada `JFrame`.

Para conseguir tener una interfaz GUI en pantalla, lo primero que tenemos que hacer es crear y visualizar un marco. El Código 11.1 muestra una clase completa (que ya se denomina `ImageViewer` en preparación de las cosas que vamos a desarrollar), que muestra un marco en pantalla. Esta clase está disponible en los proyectos del libro con el nombre *imageviewer0-1* (el número hace referencia a la versión 0.1).

Código 11.1
Una primera versión de la clase `ImageViewer`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Se omiten los comentarios.

public class ImageViewer
{
    private JFrame frame;

    /**
     * Crear un ImageViewer y mostrarlo en pantalla.
     */
    public ImageViewer()
    {
        makeFrame();
    }

    /**
     * Crear el marco Swing y su contenido.
     */
    private void makeFrame()
    {
        frame = new JFrame("ImageViewer");
        Container contentPane = frame.getContentPane();

        JLabel label = new JLabel("I am a label.");
        contentPane.add(label);

        frame.pack();
        frame.setVisible(true);
    }
}
```

Ejercicio 11.1 Abra el proyecto *imageviewer0-1*. (Esta será la base de su propio visualizador de imágenes.) Cree una instancia de la clase `ImageViewer`. Redimensione el marco resultante (hágalo más grande). ¿Qué es lo que puede observar acerca de la posición del texto dentro del marco?

Ahora vamos a analizar con cierto detalle la clase `ImageViewer` mostrada en el Código 11.1.

Las tres primeras líneas de esa clase son instrucciones para la importación de todas las clases de los paquetes `java.awt`, `java.awt.event` y `javax.swing`.[1] Necesitaremos muchas de las clases de estos paquetes para todas las aplicaciones Swing que creemos, por lo que en nuestros programas GUI siempre importaremos los tres paquetes completos.

---

[1] El paquete `swing` está realmente contenido en un paquete denominado `javax` (terminado en *x*), no `java`. La razón de esto es de carácter histórico, no parece que exista ninguna explicación lógica.

Si examinamos el resto de la clase, vemos rápidamente que todo lo interesante se encuentra dentro del método `makeFrame`. Este método se encarga de construir la GUI. El constructor de la clase contiene únicamente una llamada a este método. Hemos hecho esto para que todo el código de construcción de la GUI se encuentre en un lugar bien definido y sea fácil de encontrar posteriormente (¡cohesión!). Haremos esto mismo en todos nuestros ejemplos de interfaces GUI.

La clase tiene una variable de instancia de tipo `JFrame`. Esta variable se emplea para almacenar el marco que el visualizador de imágenes quiere mostrar en pantalla. Examinemos ahora más detenidamente el método `makeFrame`.

La primera línea de este método es

```
frame = new JFrame("ImageViewer");
```

Esta instrucción crea un nuevo marco y lo almacena en nuestra variable de instancia para su uso posterior.

Como principio general, debería examinar, mientras estudia los ejemplos de este libro, la documentación de todas las clases con que nos encontremos. Esto se aplica a todas las clases que utilicemos; en lo sucesivo, no volveremos a recordar este punto, pero esperamos que se acuerde usted de examinar la documentación.

Ejercicio 11.2 Localice la documentación de la clase `JFrame`. ¿Cuál es el propósito del parámetro "`ImageViewer`" que hemos utilizado en la llamada anterior al constructor?

Concepto:

Los componentes se colocan en un marco añadiéndolos a la **barra de menús** del marco o al **panel de contenido**.

Un marco está compuesto por tres partes: la *barra de título*, una *barra de menú* opcional y el *panel de contenido* (Figura 11.3). La apariencia exacta de la barra de título depende del sistema operativo subyacente. Normalmente, contiene el título de la ventana y unos cuantos controles de ventana.

La barra de menú y el panel de contenido están bajo el control de la aplicación. Para crear una GUI, podemos añadir algunos componentes a ambas partes del marco. Vamos a concentrarnos primero en el panel de contenido.

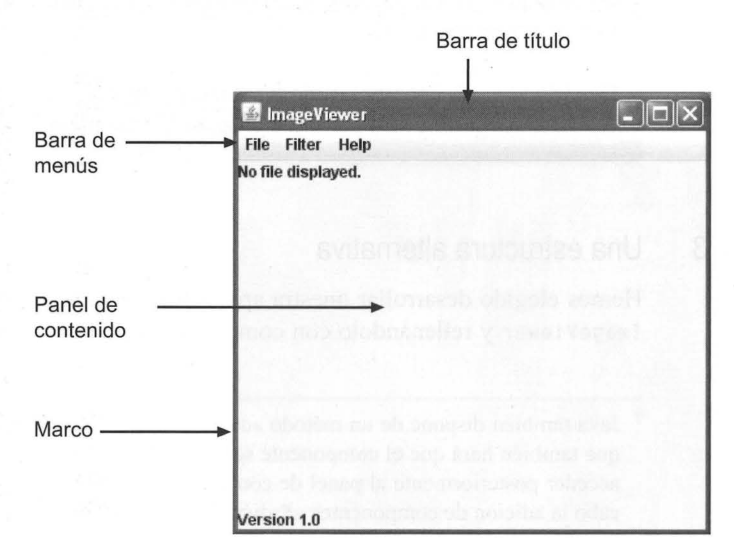


Figura 11.3
Las diferentes partes de un marco.

11.4.2 Adición de componentes simples

Inmediatamente después de la creación del JFrame, el marco será invisible y su panel de contenido estará vacío. Vamos a continuar añadiendo una etiqueta al panel de contenido:

```
Container contentPane = frame.getContentPane();

JLabel label = new JLabel("I am a label.");
contentPane.add(label);
```

La primera línea extrae una referencia al panel de contenido del marco. Siempre vamos a tener que hacer esto; los componentes de una GUI se añaden a un marco agregándolos al panel de contenido del marco.[2]

El propio panel de contenido es de tipo Container. Un contenedor es un componente Swing que puede almacenar grupos arbitrarios de otros componentes –más o menos de la misma forma que un ArrayList puede almacenar una colección arbitraria de objetos. Hablaremos más detalladamente de los contenedores más adelante.

Después creamos un componente etiqueta (tipo JLabel) y lo añadimos al panel de contenido. Una etiqueta es un componente que puede mostrar texto y/o una imagen.

Para terminar, tenemos las dos líneas

```
frame.pack();
frame.setVisible(true);
```

La primera línea hace que el marco disponga apropiadamente los componentes que contiene y que se dote asimismo del tamaño adecuado. Siempre vamos a tener que invocar el método pack sobre el marco después de añadir o redimensionar componentes.

Finalmente, la última línea hace que el marco sea visible en pantalla. Siempre vamos a comenzar con el marco invisible, para poder disponer todos los componentes en su interior sin que el proceso de construcción sea visible en pantalla. Después, cuando el marco esté ya construido, podemos mostrarlo en un estado terminado.

Ejercicio 11.3 Otro componente Swing muy utilizado es el botón (tipo JButton). Sustituya la etiqueta del ejemplo anterior por un botón.

Ejercicio 11.4 ¿Qué sucede cuando añadimos dos etiquetas (o dos botones) al panel de contenido? ¿Puede explicar lo que observe? Experimente redimensionando el marco.

11.4.3 Una estructura alternativa

Hemos elegido desarrollar nuestra aplicación creando un objeto JFrame como un atributo del ImageViewer y rellenándolo con componentes GUI adicionales, que se crean fuera del objeto

[2] Java también dispone de un método add para añadir componentes directamente en la clase JFrame, lo que también hará que el componente se añada al panel de contenido. Sin embargo, vamos a necesitar acceder posteriormente al panel de contenido de todos modos, por lo que en este punto llevaremos a cabo la adición de componentes añadiéndolos al panel de contenido.

marco. Una estructura alternativa a esta consistiría en definir `ImageViewer` como subclase de `JFrame` y rellenarlo internamente. Este estilo también es bastante común y el Código 11.2 muestra cuál sería el equivalente en este estilo al Código 11.1. Esta versión está disponible con el nombre de proyecto *imageviewer0-1a*. Aunque merece la pena estar familiarizado con ambos estilos, ninguno de los dos es necesariamente mejor que el otro, y nosotros vamos a continuar con nuestra versión original en el resto del capítulo.

Código 11.2 Una estructura alternativa para la clase `ImageViewer`.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// Comentario omitido.

public class ImageViewer extends JFrame
{
    /**
     * Crear un ImageViewer y mostrarlo en pantalla.
     */
    public ImageViewer()
    {
        super("ImageViewer");
        makeFrame();
        setVisible(true);
    }

    /**
     * Crear el marco Swing y su contenido.
     */
    private void makeFrame()
    {
        Container contentPane = getContentPane();
        JLabel label = new JLabel("I am a label.");
        contentPane.add(label);
        pack();
    }
}
```

11.4.4 Adición de menús

Nuestro siguiente paso para la construcción de una GUI consiste en añadir menús y elementos de menú. Esto es conceptualmente nuy sencillo, pero hay un detalle que tiene sus complicaciones: ¿cómo nos la arreglamos para reaccionar a las acciones del usuario, como por ejemplo la selección de un elemento de menú? Hablaremos de esto más adelante.

Primero, vamos a crear los menús. Son tres las clases implicadas:

- `JMenuBar` Un objeto de esta clase representa una barra de menús que puede mostrarse debajo de la barra de título ubicada en la parte superior de la ventana (véase la Figura 11.3). Cada ventana tiene como máximo una `JMenuBar`.[3]
- `JMenu` Los objetos de esta clase representan un único menú (como por ejemplo los habituales menús *File, Edit* o *Help –Archivo, Edición o Ayuda–*). Los menús suelen estar contenidos en una barra de menús. También pueden aparecer en los menús emergentes, pero por el momento no vamos a hacer uso de esa posibilidad.
- `JMenuItem` Los objetos de esta clase representan un único elemento de menú dentro de un menú (como por ejemplo *Open* o *Save –Abrir* o *Guardar–*).

Para nuestro visualizador de imágenes, vamos a crear una barra de menús y varios menús y elementos de menú.

La clase `JFrame` tiene un método denominado `setJMenuBar`. Podemos crear una barra de menú sy utilizar este método para asociar la barra de menús al marco:

```
JMenuBar menubar = new JMenuBar();
frame.setJMenuBar(menubar);
```

Ahora estamos listos para crear un menú y añadirlo a la barra de menús:

```
JMenu fileMenu = new JMenu("File");
menubar.add(fileMenu);
```

Estas dos líneas crean un menú etiquetado como *File* y lo insertan en la barra de menús. Por último, podemos añadir elementos de menú al menú. Las siguientes líneas añaden al menú *File* dos elementos, etiquetados como *Open* y *Quit*:

```
JMenuItem openItem = new JMenuItem("Open");
fileMenu.add(openItem);

JMenuItem quitItem = new JMenuItem("Quit");
fileMenu.add(quitItem);
```

Ejercicio 11.5 Añada el menú y los elementos de menú que acabamos de exponer a su proyecto del visualizador de imágenes. ¿Qué sucede cuando se selecciona un elemento de menú?

Ejercicio 11.6 Añada otro menú denominado *Help* que contenga un elemento de menú llamado *About ImageViewer*. (Nota: para mejorar la legibilidad y la cohesión, puede ser una buena idea mover la creación de los menús a un método separado, que quizá pudiera llamarse `makeMenuBar` y que sería invocado desde nuestro método `makeFrame`.)

[3] En Mac OS, la visualización nativa es diferente: la barra de menús se encuentra en la parte superior de la pantalla, no en la parte superior de cada ventana. En las aplicaciones Java, el comportamiento predeterminado consiste en asociar la barra de menús a la ventana. En las aplicaciones Java, puede colocarse en la parte superior de la pantalla utilizando una propiedad específica de Mac OS.

Hasta ahora, hemos conseguido completar tan solo la mitad de la tarea: podemos crear y visualizar menús, pero nos falta la otra mitad –todavía no sucede nada cuando un usuario selecciona un menú. Ahora tenemos que añadir código para reaccionar a las selecciones de menú. Este será el tema de la siguiente sección.

11.4.5 Tratamiento de sucesos

Swing utiliza un modelo muy flexible para manejar la entrada de la GUI: un modelo de *tratamiento de sucesos* con *escuchas de sucesos*.

El propio entorno de trabajo Swing y algunos de sus componentes generan sucesos cuando sucede algo en lo que puedan estar interesados otros objetos. Existen diferentes tipos de sucesos, provocados por distintos tipos de acciones. Cuando se hace clic en un botón o se selecciona un elemento de menú, el componente genera un suceso `ActionEvent`. Cuando se hace clic con el ratón o se mueve este se genera un suceso `MouseEvent`. Cuando se cierra o se reduce a un icono un marco, se genera un suceso `WindowEvent`. Existen asimismo muchos otros tipos de sucesos.

Concepto:

Un objeto puede escuchar los sucesos de los componentes implementando una interfaz de **escucha de sucesos**.

Cualquiera de nuestros objetos puede convertirse en escucha de sucesos, es decir, puede ponerse a la escucha de cualquiera de estos sucesos. Cuando se pone a la escucha, recibirá una notificación por cada uno de los sucesos que esté escuchando. Un objeto se convierte en un escucha de sucesos implementando una de las diversas interfaces de escucha existentes. Si implementa la interfaz correcta, puede registrarse ante un componente del cual quiera quedarse a la escucha.

Veamos un ejemplo. Un elemento de menú (clase `JMenuItem`) genera un suceso `ActionEvent` al ser activado por un usuario. Los objetos que quieran escuchar esos sucesos tienen que implementar la interfaz `ActionListener` del paquete `java.awt.event`.

Hay dos estilos alternativos para la implementación de escuchas de sucesos: o bien un único objeto escucha los sucesos de muchas fuentes de sucesos distintas, o bien a cada una de las fuentes de sucesos se le asigna su propio escucha. Hablaremos de ambos estilos en las dos secciones siguientes.

11.4.6 Recepción centralizada de sucesos

Para hacer que nuestro objeto `ImageViewer` sea el único escucha de todos los sucesos del menú, tenemos que hacer tres cosas:

1. Debemos declarar en la cabecera de la clase que implementa la interfaz `ActionListener`.
2. Tenemos que implementar un método con la signatura

   ```
   public void actionPerformed(ActionEvent e)
   ```

   Este es el único método declarado en la interfaz `ActionListener`.
3. Tenemos que invocar el método `addActionListener` del elemento de menú, para registrar cómo escucha el objeto `ImageViewer`.

Los puntos 1 y 2 (implementar la interfaz y definir su método) garantizan que nuestro objeto sea un subtipo de `ActionListener`. El punto 3 registra nuestro propio objeto como escucha de los elementos de menú. El Código 11.3 muestra el código fuente para hacer esto en nuestro contexto.

Código 11.3
Adición de un escucha de acción a un elemento de menú.

```
public class ImageViewer
    implements ActionListener
{

    // Se omiten los campos y el constructor.

    public void actionPerformed(ActionEvent event)
    {
        System.out.println("Menu item: " +
                           event.getActionCommand());
    }

    /**
     * Crear el marco Swing y su contenido.
     */
    private void makeFrame()
    {
        frame = new JFrame("ImageViewer");
        makeMenuBar(frame);

        // Se omite parte del código de construcción de la GUI.
    }

    /**
     * Crear la barra de menús del
     * marco principal.
     * @param frame El marco al que hay que añadir la barra
     *              de menús
     */
    private void makeMenuBar(JFrame frame)
    {

        JMenuBar menubar = new JMenuBar();
        frame.setJMenuBar(menubar);
        // crear el menú File
        JMenu fileMenu = new JMenu("File");
        menubar.add(fileMenu);

        JMenuItem openItem = new JMenuItem("Open");
        openItem.addActionListener(this);
        fileMenu.add(openItem);

        JMenuItem quitItem = new JMenuItem("Quit");
        quitItem.addActionListener(this);
        fileMenu.add(quitItem);
    }
}
```

En el ejemplo de código anterior, fíjese especialmente en las líneas

```
JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(this);
```

En ellas, se crea un elemento de menú y se registra el objeto actual (el propio objeto `ImageViewer`) como un escucha de acción, pasando `this` como parámetro al método `addActionListener`.

El efecto de registrar nuestro objeto como escucha ante el elemento de menú es que el elemento de menú invocará nuestro propio método `actionPerformed` cada vez que se active el elemento. Cuando se invoque nuestro método, el elemento de menú le pasará un parámetro de tipo `ActionEvent` que proporcionará algunos detalles acerca del suceso que ha tenido lugar. Estos detalles incluyen el instante exacto del suceso, el estado de las teclas modificadoras (las teclas Mayús, Ctrl y meta), una "cadena de comandos" y otros detalles.

La cadena de comandos es una cadena que identifica de alguna manera el componente que ha provocado el suceso. Para los elementos de menú, esta será, de forma predeterminada, el texto de la etiqueta del elemento.

En nuestro ejemplo del Código 11.3, registramos el mismo objeto de acción para ambos elementos de menú. Esto significa que ambos elementos de menú invocarán el mismo método `actionPerformed` cuando sean activados.

En el método `actionPerformed`, simplemente imprimimos la cadena de comando del elemento, para demostrar que este esquema funciona. Obviamente, podemos ahora añadir código para gestionar apropiadamente la invocación del menú.

Este ejemplo de código, tal como lo hemos explicado hasta el momento, está disponible en los proyectos del libro, con el nombre de proyecto *imageviewer0-2*.

Ejercicio 11.7 Implemente el código de gestión de menú que acabamos de explicar en su propio proyecto del visualizador de imágenes. Alternativamente, abra el proyecto *imageviewer0-2* y examine cuidadosamente el código fuente. Describa por escrito y detalladamente la secuencia de sucesos que se produce al activar el elemento de menú *Quit*.

Ejercicio 11.8 Añada otro elemento de menú denominado *Save*.

Ejercicio 11.9 Añada tres métodos privados a su clase, denominados `openFile`, `saveFile` y `quit`. Modifique el método `actionPerformed` de modo que invoque el método correspondiente cuando se active un elemento de menú.

Ejercicio 11.10 Si ya hecho el Ejercicio 11.6 (añadir un menú *Help*), asegúrese de que también se gestione apropiadamente su elemento de menú.

Podemos comprobar que este enfoque funciona, e implementar a continuación métodos para gestionar los elementos de menú, con el fin de llevar a cabo las diversas tareas del programa. Existe, sin embargo, otro aspecto que debemos investigar: la solución actual no es muy elegante en términos de ampliabilidad y mantenibilidad.

Examine el código que ha escrito en el método `actionPerformed` para resolver el Ejercicio 11.9. Hay varios problemas:

- Probablemente ha utilizado una instrucción if y el método `getActionCommand` para averiguar qué elemento fue activado. Por ejemplo, se podría escribir

  ```
  if(event.getActionCommand().equals("Open")) ...
  ```

  Depender de la cadena de caracteres que forma la etiqueta del elemento para realizar la función no es una buena idea. ¿Qué pasaría si ahora tradujéramos la interfaz a otro idioma? Con solo cambiar el texto del elemento del menú, el programa dejaría de funcionar (o bien tendríamos que localizar todos los lugares en el código donde se ha utilizado esa cadena de caracteres y cambiarla –un procedimiento tedioso y proclive a errores).

- Disponer de un método de despacho central (como nuestro `actionPerformed`) no proporciona una estructura elegante. Básicamente, lo que hacemos es que todos los elementos llamen a un mismo método, tan solo para a continuación escribir un tedioso código en este método que invoque diferentes métodos para cada elemento. Esto es muy molesto en términos de mantenimiento (para cada elemento de menú adicional, tendríamos que añadir una nueva instrucción if en `actionPerformed`); también es un desperdicio de esfuerzos. Sería mucho más conveniente que pudiéramos hacer que cada elemento de menú invocara directamente un método distinto.

En la siguiente sección, vamos a presentar una nueva estructura del lenguaje que nos permite hacer precisamente eso.

11.4.7 Clases internas

Para resolver el problema que acabamos de mencionar relativo al despacho centralizado de sucesos, utilizaremos una nueva estructura que aun no hemos presentado: las *clases internas*. Las clases internas son clases que se declaran textualmente dentro de otra clase:

```
class ClaseCircundante
{
    ...
    class ClaseInterna
    {
        ...
    }
}
```

Las instancias de la clase interna están asociadas a instancias de la clase circundante; solo pueden existir junto con una instancia circundante, y existen conceptualmente *dentro* de la instancia circundante. Un detalle interesante es que las instrucciones contenidas en los métodos de la clase interna pueden ver los campos y métodos privados de la clase circundante y acceder a ellos. Por tanto, existe, obviamente, un acoplamiento muy estrecho entre las dos clases. La clase interna se considera una parte de la clase circundante, de la misma forma que cualquiera de los métodos contenidos en la clase circundante.

Ahora podemos utilizar esta estructura para definir una clase separada de escucha de acción para cada elemento de menú que queramos escuchar. Como se trata de clases separadas, cada una puede disponer de un método `actionPerformed` distinto, de modo que cada uno de esos métodos gestionará la activación de un único elemento. La estructura es la siguiente:

```
class ImageViewer
{
    . . .
    class OpenActionListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            // realizar la acción correspondiente a open
        }
    }
    class QuitActionListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            // realizar la acción correspondiente a quit
        }
    }
}
```

(Como guía de estilo, solemos escribir las clases internas al final de la clase circundante, después de los métodos.)

Una vez que hemos hecho esto, podemos ahora crear instancias de estas clases internas exactamente de la misma forma que lo haríamos para cualquier otra clase. Observe también que `ImageViewer` no implementa ya `ActionListener` (eliminamos su método `actionPerformed`), pero las dos clases internas sí lo hacen. Esto nos permite ahora utilizar instancias de las clases internas como escuchas de acción para los elementos de menú.

```
JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(new OpenActionListener());
. . .
JMenuItem quitItem = new JMenuItem("Quit");
quitItem.addActionListener(new QuitActionListener());
```

En resumen, en lugar de hacer que el objeto visualizador de imágenes escuche todos los sucesos de acción, hemos creado objetos escucha separados para cada posible suceso, dedicando cada uno de ellos a escuchar un único tipo de suceso. Puesto que todo escucha tiene su propio método `actionPerformed`, ahora podemos escribir en esos métodos el código específico para el tratamiento del suceso que deseemos. Asimismo, puesto que las clases escucha se encuentran dentro del ámbito de la clase circundante (pueden acceder a los campos y métodos privados de la clase circundante), pueden hacer un uso completo de la clase circundante en la implementación de los métodos `actionPerformed`.

Observe un par de características de estos objetos escucha:

- No nos preocupamos de almacenarlos en variables –por tanto, en la práctica, se trata de objetos anónimos. Solo los elementos de menú disponen de una referencia a los objetos escucha, para poder invocar sus métodos `actionPerformed`.
- Creamos un único objeto para cada una de las clases internas, ya que cada clase está altamente especializada para un elemento de menú concreto.

Estas características nos llevarán a explorar otra característica adicional de Java en la siguiente sección.

Ejercicio 11.11 Implemente la gestión de los elementos de menú con clases internas, como hemos explicado aquí, en su propia versión del visualizador de imágenes.

En algunos casos, las clases internas pueden utilizarse con carácter general para mejorar la cohesión en proyectos de gran tamaño. El proyecto *foxes-and-rabbits* del Capítulo 10, por ejemplo, tiene una clase `SimulatorView` que incluye una clase interna, `FieldView`. Pruebe a estudiar ese ejemplo para mejorar su comprensión del concepto de clase interna.

11.4.8 Clases internas anónimas

La solución al problema del despacho de acciones mediante clases internas es bastante buena, pero nos gustaría llevarla un paso más allá: podemos usar *clases internas anónimas*. El proyecto *imageviewer0-3* muestra una implementación utilizando esta estructura.

Ejercicio 11.12 Abra el proyecto *imageviewer0-3* y examínelo; es decir, pruébelo y lea su código fuente. No se preocupe si no entiende todos los aspectos, porque algunas nuevas características son precisamente el tema de esta sección. ¿Qué es lo que observa acerca del uso de clases internas a la hora de permitir que `ImageViewer` escuche los sucesos y los trate?

Ejercicio 11.13 Observará que al activar el elemento de menú *Quit*, salimos del programa. Examine cómo se hace esto. Busque la documentación de la librería para las clases y métodos implicados.

Uno de los aspectos fundamentales de los cambios incluidos en esta versión es la forma en que se configuran los escuchas de acción para que escuchen los sucesos de acción de los elementos de menú. El código relevante tiene el siguiente aspecto:

```
JMenuItem openItem = new JMenuItem("Open");
openItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) { openFile(); }
       });
```

Este fragmento de código parece bastante misterioso cuando nos encontramos con él por primera vez, y probablemente tenga problemas en interpretarlo aunque haya comprendido todo lo que hemos explicado en el libro hasta ahora. Esta estructura es, probablemente, el ejemplo más confuso desde el punto de vista sintáctico que jamás se encontrará en el lenguaje Java. Pero no se preocupe, vamos a analizar detalladamente esa estructura.

Lo que podemos ver es una clase interna anónima. La idea de esta estructura está basada en las observaciones efectuadas en la versión anterior, donde vimos que solo utilizamos cada clase interna exactamente una vez para crear una única instancia sin nombre. Para esta situación, las clases internas anónimas proporcionan un atajo sintáctico: nos permiten definir una clase y crear una única instancia de esa clase, todo en un único paso. El efecto es idéntico a la versión con clases internas anterior, con la diferencia de que no necesitamos definir clases con nombre separadas para los escuchas; asimismo, la definición del método escucha está situada más próxima al registro del escucha ante el elemento de menú.

El coloreado del ámbito en el editor de BlueJ nos proporciona algunas indicaciones que nos pueden ayudar a comprender esta estructura (Figura 11.4). El sombreado verde (gris más oscuro en la figura) indica una clase, el de color amarillo (gris más claro en la figura) muestra una definición de método y el fondo blanco identifica un cuerpo de método. Podemos ver que el cuerpo del método makeMenuBar contiene, muy densamente empaquetadas, dos definiciones de clase (de aspecto extraño), cada una de las cuales tiene una única definición de método dentro de un cuerpo de corta longitud.

Figura 11.4
Coloreado de ámbitos con dos clases internas anónimas.

```
/**
 * Create the main frame's menu bar.
 * @param frame   The frame that the menu bar should be added to.
 */
private void makeMenuBar(JFrame frame)
{
    JMenuBar menubar = new JMenuBar();
    frame.setJMenuBar(menubar);

    // create the File menu
    JMenu fileMenu = new JMenu("File");
    menubar.add(fileMenu);

    JMenuItem openItem = new JMenuItem("Open");
        openItem.addActionListener(new ActionListener() {
                               public void actionPerformed(ActionEvent e) { openFile(); }
                           });
    fileMenu.add(openItem);

    JMenuItem quitItem = new JMenuItem("Quit");
        quitItem.addActionListener(new ActionListener() {
                               public void actionPerformed(ActionEvent e) { quit(); }
                           });
    fileMenu.add(quitItem);
}
}
```

Cuando utilizamos una clase interna anónima, creamos una clase interna *sin asignarla un nombre* e inmediatamente creamos una única instancia de esa clase. En el código del escucha de acción anterior, esto se hace mediante el fragmento de código:

```
new ActionListener() {
        public void actionPerformed(ActionEvent e) { openFile(); }
    }
```

La manera de crear una clase interna anónima es mediante el nombre de un supertipo (que a menudo suele ser un clase abstracta o una interfaz –aquí ActionListener), seguido de un bloque que contiene una implementación para sus métodos abstractos. Esto tiene un aspecto inusual, porque no hay ningún otro caso en el que se permita crear directamente una instancia de una clase abstracta o interfaz.

En este ejemplo, hemos creado un nuevo subtipo de ActionListener que implementa el método actionPerformed. Esta nueva clase no recibe ningún nombre. En lugar de ello, le anteponemos la palabra clave new para crear una única instancia de dicha clase.

En nuestro ejemplo, esta única instancia es un objeto escucha de acción (es de un subtipo de ActionListener). Se puede pasar al método addActionListener de un elemento de menú y este invocará el método openFile de su clase circundante al ser activado. Cada subtipo de ActionListener creado de esta forma representa una clase anónima distinta.

Al igual que las clases internas con nombre, las clases internas anónimas pueden acceder a los campos y métodos de su clase circundante. Además, puesto que están definidas dentro de un método, pueden acceder a las variables locales y parámetros de dicho método. Sin embargo, una regla importante es que las variables locales a las que se acceda de esta forma deben estar declaradas como variables `final`. Puede ver un ejemplo de esto en el proyecto *imageviewer2-0* del que hablaremos en la Sección 11.6.

Merece la pena recalcar algunas observaciones sobre las clases internas anónimas. En primer lugar, para nuestro problema concreto, resulta muy útil emplear clases internas anónimas. Nos permiten eliminar completamente el método `actionPerformed` central de nuestra clase `ImageViewer`. Y lo que hacemos en lugar de ello es crear un escucha de acción (clase y objeto) independiente y personalizado para cada elemento de menú. Este escucha de acción puede invocar directamente el método que implementa la correspondiente función.

Esta estructura tiene un alto grado de cohesión y ampliabilidad. Si necesitamos un elemento de menú adicional, nos basta con añadir el código para crear el elemento y su escucha, así como el método que gestiona su función. No hace falta ninguna enumeración en ningún método central.

Concepto:

Las **clases internas anónimas** son una estructura muy útil a la hora de implementar escuchas de sucesos.

En segundo lugar, la utilización de clases internas anónimas puede hacer que el código sea bastante difícil de leer. Se recomienda vivamente utilizarlas tan solo para clases muy cortas y para estructuras de código muy comunes. En lo que a nosotros respecta, el único ejemplo de este libro donde vamos a usar las clases anónimas es para la implementación de los escuchas de sucesos.[4]

En tercer lugar, a menudo utilizamos clases anónimas cuando solo se va a requerir una única instancia de la implementación –cuando las acciones asociadas con cada elemento de menú son características de ese elemento concreto. Además, todas las referencias a la instancia serán siempre a través de su supertipo. Ambas razones implican que existe una menor necesidad de dotar de un nombre a la nueva clase; de aquí que pueda ser anónima.

En nuestro análisis posterior, vamos a evitar emplear el método `actionPerformed` central y usaremos en su lugar clases internas anónimas. Por tanto, olvídese el proyecto *imageviewer0-2* y utilice la estructura de *imageviewer0-3* como base para sus próximas implementaciones.

11.4.9 Resumen de los elementos clave de una GUI

Al principio de este capítulo, hemos enumerado las tres áreas importantes en la construcción de una GUI: componentes, diseño gráfico y tratamiento de sucesos. Hasta ahora, nos hemos concentrado en dos de estas áreas. Hemos hablado de un pequeño número de componentes (etiquetas, botones, menús, elementos de menú) y hemos explicado con un cierto grado de detalle cómo se gestionan los sucesos de acción.

Llegar hasta aquí (y ser capaces de mostrar un marco con una etiqueta y unos pocos menús) ha representado un gran trabajo, y hemos tenido que hablar de un montón de conceptos fundamentales. A partir de ahora, las cosas van a ser algo más fáciles. Comprender el

---

[4] Si desea obtener más información acerca de las clases internas, eche un vistazo a estas dos secciones del tutorial Java en línea: `http://download.oracle.com/javase/tutorial/java/javaOO/nested.html` y `http://download.oracle.com/javase/tutorial/java/javaOO/innerclasses.html`.

tratamiento de sucesos para los elementos de menú ha sido, probablemente, el detalle más difícil a la hora de analizar nuestro ejemplo.

Añadir más menús y otros componentes al marco será ahora muy sencillo –en esencia, más de lo mismo. El área a la que echaremos un vistazo y que sí es completamente nueva es la del *diseño gráfico*: cómo colocar los componentes dentro del marco.

11.5 ImageViewer 1.0: la primera versión completa

Ahora vamos a trabajar en la creación de la primera versión completa, una que realmente pueda llevar a cabo la tarea principal: mostrar algunas imágenes.

11.5.1 Clases de procesamiento de imágenes

Como paso intermedio para la solución vamos a investigar otra versión provisional: *imageviewer0-4*. Su estructura de clases se muestra en la Figura 11.5.

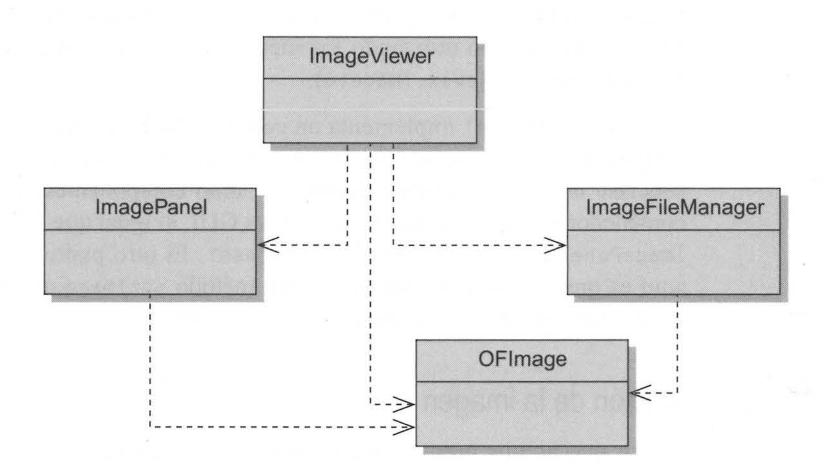


Figura 11.5
La estructura de clases de la aplicación de visualización de imágenes.

Como puede ver, hemos añadido tres nuevas clases: `OFImage`, `ImagePanel` y `ImageFileManager`. `OFImage` es una clase que sirve para representar una imagen que queramos visualizar y manipular. `ImageFileManager` es una clase auxiliar que proporciona métodos estáticos para leer un archivo de imagen (en formato JPEG o PNG) de disco y devolverlo en formato `OFImage` y luego guardar de nuevo en disco el objeto `OFImage`. `ImagePanel` es un componente Swing personalizado para mostrar la imagen en nuestra GUI.

Vamos a analizar brevemente los aspectos más importantes de cada una de estas clases con algo más de detalle. Sin embargo, no las vamos a explicar completamente –eso lo dejamos como ejercicio para el lector curioso.

La clase `OFImage` es nuestro propio formato personalizado para representar una imagen en memoria. Podemos pensar en `OFImage` como en una matriz bidimensional de píxeles. Cada uno de los píxeles puede tener un color. Utilizamos la clase estándar `Color` (del paquete `java.awt`) para representar el color de cada píxel. Eche un vistazo también a la documentación de la clase `Color`; la necesitaremos más adelante.

OFImage está implementada como subclase de la clase estándar Java BufferedImage (del paquete java.awt.image). BufferedImage nos proporciona la mayor parte de la funcionalidad que necesitamos (también representa una imagen como una matriz bidimensional), pero no dispone de métodos para configurar o consultar un píxel utilizando un objeto Color (utiliza diferentes formatos para esto que no queremos emplear). Por tanto, hemos definido nuestra propia subclase, que añade estos dos métodos.

Para este proyecto, podemos tratar OFImage como una clase de librería; no necesitamos modificar dicha clase.

Los métodos más importantes de OFImage para nosotros son:

- getPixel y setPixel para leer y modificar píxeles de uno en uno.
- getHeight y getWidth para determinar el tamaño de la imagen.

La clase ImageFileManager ofrece tres métodos: uno para leer un archivo de imagen con nombre desde disco y devolverlo como un objeto OFImage, otro para escribir un archivo OFImage en disco y otro para abrir un cuadro de diálogo de selección de archivo, para que el usuario pueda seleccionar la imagen que desea abrir. Los métodos pueden leer archivos en los formatos JPEG y PNG estándar, y el método save escribirá los archivos en formato JPEG. Esto se lleva a cabo utilizando los método de E/S de imágenes estándar de Java, de la clase ImageIO (paquete javax.imageio).

La clase ImagePanel implementa un componente Swing personalizado para visualizar nuestra imagen. Los componentes Swing personalizados pueden crearse fácilmente escribiendo una subclase de un componente existente. Como componentes que son pueden insertarse en un contenedor Swing y mostrarse en nuestra GUI, al igual que cualquier otro componente Swing. ImagePanel es una subclase de JComponent. El otro punto importante que hay que destacar aquí es que ImagePanel dispone de un método setImage que toma un objeto OFImage como parámetro, con el fin de visualizar cualquier objeto OFImage que queramos.

11.5.2 Adición de la imagen

Ahora que hemos preparado las clases para manejar las imágenes, resulta sencillo añadir la imagen a la interfaz de usuario. El Código 11.4 muestra las diferencias importantes con respecto a las versiones anteriores.

Al comparar este código con la versión anterior, observamos que solo hay dos pequeños cambios:

- En el método makeFrame, ahora creamos y añadimos un componente ImagePanel en lugar de un JLabel. Hacer esto no es más complicado que añadir la etiqueta. El objeto ImagePanel se almacena en un campo de instancia, para que podamos acceder a él de nuevo posteriormente.
- Hemos modificado nuestro método openFile para que abra y muestre un archivo de imagen. Utilizando nuestras clases para el procesamiento de imágenes, esta tarea también resulta sencilla. La clase ImageFileManager tiene un método para seleccionar y abrir una imagen. y el objeto ImagePanel dispone de un método para visualizar esa imagen. Un aspecto que hay que resaltar es que necesitamos invocar frame.pack() al final del método openFile, ya que el tamaño de nuestro componente de imagen habrá cambiado. El método pack recalculará la colocación del marco y lo redibujará para que se gestione adecuadamente el cambio de tamaño.

Código 11.4
La clase `ImageViewer` con `ImagePanel`.

```
public class ImageViewer
{
    private JFrame frame;
    private ImagePanel imagePanel;

    // Se omiten el constructor y el método para salir de la aplicación.

    /**
     * Función Open: abre un selector de archivos para elegir
     * un nuevo archivo de imagen.
     */
    private void openFile()
    {
        OFImage image = ImageFileManager.getImage();
        imagePanel.setImage(image);
        frame.pack();
    }

    /**
     * Crear el marco Swing y su contenido.
     */
    private void makeFrame()
    {
        frame = new JFrame("ImageViewer");
        makeMenuBar(frame);

        Container contentPane = frame.getContentPane();

        imagePanel = new ImagePanel();
        contentPane.add(imagePanel);

        // terminada la construcción - colocar los componentes
        // y mostrar
        frame.pack();
        frame.setVisible(true);
    }

    // Método MakeMenuBar omitido.
}
```

Ejercicio 11.14 Abra y pruebe el proyecto *imageviewer0-4*. La carpeta para los proyectos de este capítulo incluye una carpeta denominada *images*. En ella podrá encontrar algunas imágenes de prueba que puede utilizar, aunque también puede, por supuesto, emplear sus propias imágenes.

Ejercicio 11.15 ¿Qué sucede cuando abrimos una imagen y luego redimensionamos el marco? ¿Qué sucede si redimensionamos primero el marco y luego abrimos una imagen?

Con esta versión, hemos resuelto la tarea central: ahora podemos abrir un archivo de imagen almacenado en el disco y mostrarlo en pantalla. Sin embargo, antes de llamar a nuestro proyecto "versión 1.0" y declararlo finalizado por primera vez, queremos añadir unas cuantas mejoras más (véase la Figura 11.2).

- Queremos añadir dos etiquetas: una para mostrar el nombre del archivo de imagen en la parte superior y un texto de estado en la parte inferior.
- Queremos añadir un menú *Filter* que contenga algunos filtros para modificar la apariencia de la imagen.
- Queremos añadir un menú *Help* que contenga un elemento *About ImageViewer*. Al seleccionar este elemento de menú, debe mostrarse un cuadro de diálogo con el nombre de la aplicación, el número de versión e información acerca del autor.

11.5.3 Diseño gráfico

En primer lugar, trabajaremos en la tarea de añadir dos etiquetas de texto a nuestra interfaz: una en la parte superior, que se utilizará para visualizar el nombre del archivo de la imagen que se esté mostrando, y otra en la parte inferior, que se empleará para diversos mensajes de estado.

La creación de estas etiquetas es sencilla –se trata simplemente de instancias `JLabel`. Las almacenaremos en campos de instancia, para poder acceder posteriormente a ellas con el fin de modificar el texto que muestran. La única cuestión pendiente es cómo colocarlas en la pantalla.

Un primer intento (simplista e incorrecto) podría ser así:

```
Container contentPane = frame.getContentPane();

filenameLabel = new JLabel();
contentPane.add(filenameLabel);

imagePanel = new ImagePanel();
contentPane.add(imagePanel);

statusLabel = new JLabel("Version 1.0");
contentPane.add(statusLabel);
```

La idea aquí es muy simple: obtenemos el panel de contenido del marco y añadimos, uno tras otro, los tres componentes que queremos visualizar. El único problema es que no hemos especificado exactamente cómo queremos colocar esos tres componentes. Puede que queramos que aparezcan uno a continuación del otro, o uno debajo del otro o en cualquier otra posible disposición. Como no hemos especificado ningún diseño gráfico, el contenedor (el panel de contenido) recurrirá a un comportamiento predeterminado. Y resulta que esto no es lo que deseamos.

Swing utiliza *gestores de diseño gráfico* para colocar los componentes en una GUI. Cada contenedor que alberga componentes, como el panel de contenido, tiene un gestor de diseño gráfico asociado que se encarga de colocar los componentes dentro de ese contenedor.

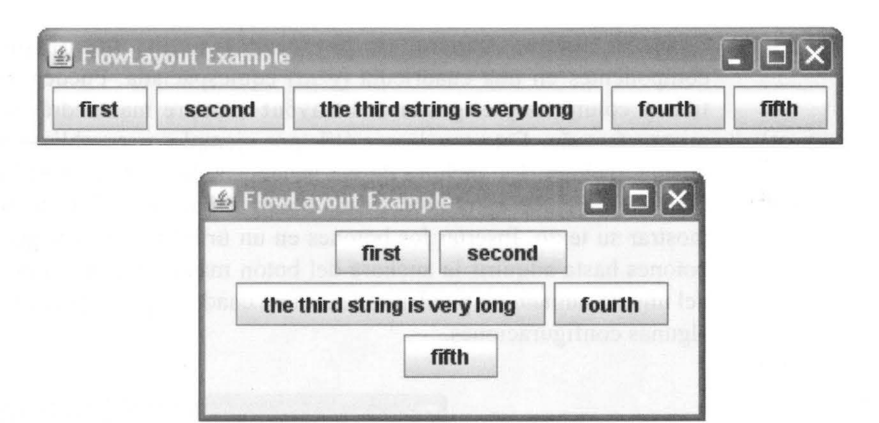
Ejercicio 11.16 Continuando con su última versión del proyecto, utilice el fragmento de código mostrado anteriormente para añadir las dos etiquetas. Pruebe esta solución. ¿Qué es lo que observa?

Swing proporciona varios gestores de diseño diferentes, para dar soporte a las distintas preferencias de colocación. Los más importantes son: `FlowLayout`, `BorderLayout`, `GridLayout` y `BoxLayout`. Cada uno de ellos está representado por una clase Java en la librería Swing y cada uno coloca de diferentes maneras los componentes que tiene bajo su control.

He aquí un breve descripción de cada uno de esos posibles diseños. Las diferencias principales entre ellos son la forma en la que se colocan los componentes y cómo se distribuye el espacio disponible entre los mismos. Puede encontrar los ejemplos ilustrados aquí en el proyecto *layouts*.

`FlowLayout` (Figura 11.6) dispone todos los componentes secuencialmente de izquierda a derecha. Dejará cada componente con su tamaño preferido y los centrará horizontalmente. Si el espacio horizontal no es suficiente como para encajar todos los componentes, algunos de ellos saltarán a una segunda línea. El gestor `FlowLayout` puede también configurarse para alinear los componentes a la izquierda o a la derecha. Dado que los componentes no se redimensionan para rellenar el espacio disponible, habrá espacio libre alrededor de ellos si se cambia el tamaño de la ventana.

Figura 11.6
`FlowLayout`.



Un `BorderLayout` (Figura 11.7) coloca hasta cinco componentes en un patrón en forma de cruz: uno en el centro y los otros cuatro componentes en las partes superior, inferior, derecha e izquierda. Cada de estas posiciones puede estar vacía, por lo que podría en total contener menos de cinco componentes. Las cinco posiciones se denominan CENTER, NORTH, SOUTH, EAST y WEST. Con un gestor `BorderLayout` no se deja ningún espacio vacío al cambiar el tamaño de la ventana; todo el espacio se distribuye (de manera no equitativa) entre los componentes.

Este diseño gráfico puede parecer bastante especializado a primera vista, lo que nos hace preguntarnos si se suele utilizar muy a menudo. Pero en la práctica es un diseño gráfico sorprendentemente útil que se utiliza en muchas aplicaciones. En BlueJ, por ejemplo, tanto la ventana principal como el editor utilizan un `BorderLayout` como gestor de diseño gráfico principal.

Cuando se redimensiona un `BorderLayout`, el componente central es el que se estira o encoge en ambas direcciones. Los componentes situados al este y al oeste cambian de altura, pero mantienen su anchura. Los componentes situados en las posiciones al norte y al sur mantienen su altura, por lo que solo varía su anchura.

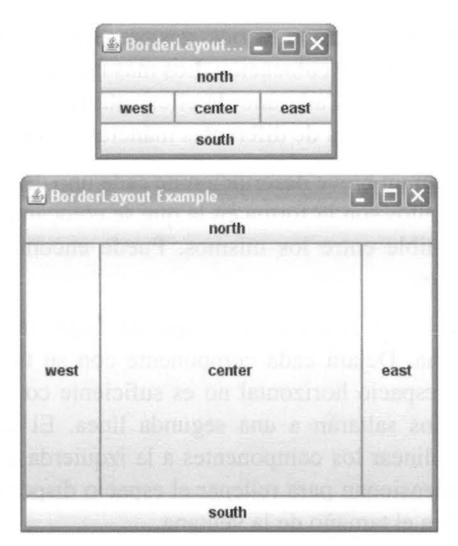


Figura 11.7
BorderLayout.

Como el nombre sugiere, un GridLayout resulta útil (Figura 11.8) para disponer los componentes en una cuadrícula (*grid*) equiespaciada. Pueden especificarse los números de filas y columnas y el gestor GridLayout siempre mantendrá todos los componentes con el mismo tamaño. Esto puede ser útil, por ejemplo, para obligar a que los botones tengan la misma anchura. La anchura de las instancias de JButton está inicialmente determinada por el texto contenido en el botón, cada botón se hace lo suficientemente ancho como para poder mostrar su texto. Insertar los botones en un GridLayout hará que se redimensionen todos los botones hasta adquirir la anchura del botón más ancho. Si un número impar de componentes del mismo tamaño no pueden rellenar una cuadrícula 2D, puede haber algo de espacio libre en algunas configuraciones.

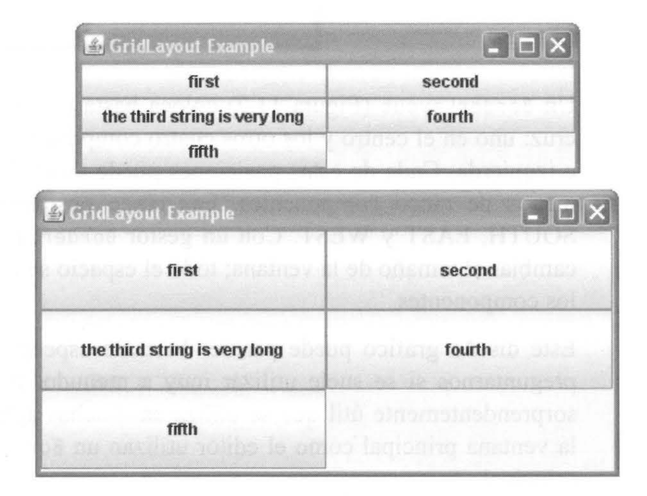


Figura 11.8
GridLayout.

Un BoxLayout coloca múltiples componentes en forma vertical u horizontal. Los componentes no se redimensionan y el gestor no hará que salten de línea o de columna al cambiar el tamaño de la ventana (Figura 11.9). Anidando varios gestores BoxLayout pueden construirse diseños gráficos sofisticados, alineados bidimensionalmente.

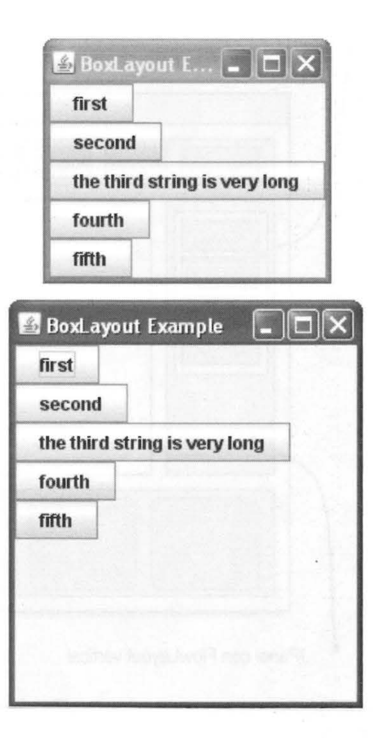


Figura 11.9
`BoxLayout`.

Ejercicio 11.17 Utilizando el proyecto *layouts* de este capítulo, experimente con los ejemplos ilustrados en esta sección. Añada y elimine componentes de las clases existentes, para ver cuáles son las características claves de los diferentes estilos de diseño gráfico. ¿Qué sucede, por ejemplo, si no hay ningún componente CENTER con `BorderLayout`?

11.5.4 Contenedores anidados

Todas las estrategias de diseño gráfico que acabamos de explicar son bastante simples. La clave para construir interfaces con un buen aspecto y un comportamiento correcto radica en un último detalle: los gestores de diseño pueden anidarse. Muchos de los componentes de Swing son *contenedores*. Los contenedores parecen ser de cara al exterior un único componente, pero pueden contener otros muchos componentes. Cada contenedor tiene asociado su propio gestor de diseño gráfico.

El contenedor más utilizado es la clase `JPanel`. Puede insertarse un `JPanel` como componente en el panel de contenido del marco, después de lo cual pueden colocarse más componentes dentro del `JPanel`. Por ejemplo, la Figura 11.10 muestra una disposición de interfaz similar a la de la ventana principal de BlueJ. El panel de contenido de este marco utiliza un `BorderLayout`, en el que la posición EAST no está utilizada. El área NORTH de este `BorderLayout` contiene un `JPanel` con un `FlowLayout` horizontal que dispone sus componentes (por ejemplo, botones de la barra de tareas) en una fila. El área SOUTH es similar: otro `JPanel` con un `FlowLayout`.

El grupo de botones en el área WEST se coloca primero en un `JPanel` con un `GridLayout` de una sola columna, para proporcionar a todos los botones la misma anchura. Este `JPanel` fue entonces colocado dentro de otro `JPanel` con un `FlowLayout` de modo que la cuadrícula no

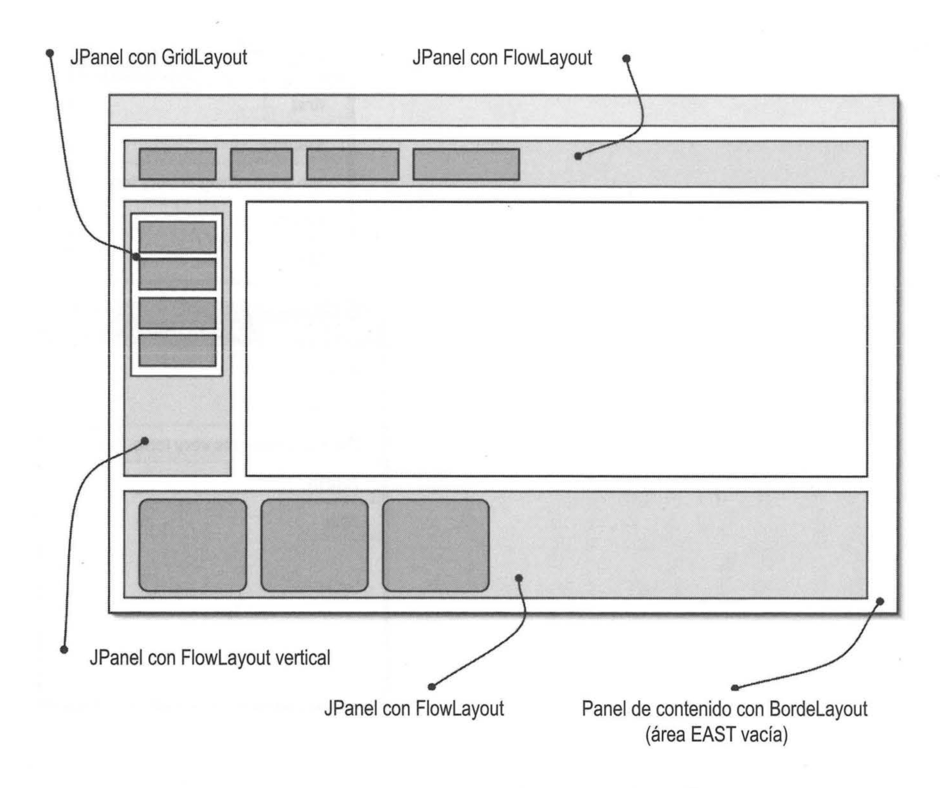


Figura 11.10 Construcción de una interfaz utilizando contenedores anidados.

ocupara toda la altura del área WEST. El `JPanel` externo se insertó entonces dentro del área WEST del marco.

Observe cómo cooperan el contenedor y el gestor de diseño gráfico a la hora de colocar los componentes. El contenedor almacena los componentes, pero es el gestor de diseño el que decide su posición exacta en la pantalla. Cada contenedor tiene un gestor de diseño gráfico; si no configuramos explícitamente uno, utilizará un gestor de diseño predeterminado. El gestor predeterminado es distinto para los diferentes contenedores: el panel de contenido de un `JFrame`, por ejemplo, tiene de forma predeterminada un `BorderLayout`, mientras que `JPanels` utiliza un `FlowLayout` por omisión.

Ejercicio 11.18 Examine la GUI del proyecto de la calculadora utilizado en el Capítulo 7 (Figura 7.6). ¿Qué tipo de contenedores/gestores de diseño gráfico cree que se utilizaron para crearlo? Después de responder por escrito, abra el proyecto *calculator-gui* y compruebe su respuesta leyendo el código.

Ejercicio 11.19 ¿Qué tipo de gestores de diseño gráfico podrían haberse utilizado para crear el diseño de la ventana del editor de BlueJ?

Ejercicio 11.20 En BlueJ, invoque la función *Use Library Class* (Utilizar clase de librería) del menú *Tools*. Examine el cuadro diálogo que aparece en pantalla. ¿Qué contenedores/gestores de diseño gráfico se han utilizado para crearlo? Redimensione el cuadro de diálogo y observe el comportamiento del mismo ante el cambio de tamaño, para obtener información adicional.

Es el momento de examinar algo de código para nuestra aplicación `ImageViewer`. Nuestro objetivo es muy simple. Queremos ver tres componentes uno encima de otro: una etiqueta

en la parte superior, la imagen en la parte central y otra etiqueta en la parte inferior. Son varios los gestores de diseño gráfico que permiten hacer esto y estará más claro cuál debemos elegir después de pensar acerca del comportamiento en caso de redimensionamiento. Cuando agrandemos la ventana, nos gustaría que las etiquetas mantuvieran su anchura y que todo el espacio adicional se asignara a la imagen. Esto sugiere un `BorderLayout`: las etiquetas pueden estar en las áreas NORTH y SOUTH, y la imagen en el área CENTER. El Código 11.5 muestra el código fuente para implementar esto.

Merece la pena resaltar dos detalles. En primer lugar, se utiliza el método `setLayout` en el panel de contenido para configurar el gestor de diseño gráfico deseado.[5] El propio gestor de diseño gráfico es un objeto, así que creamos una instancia de `BorderLayout` y lo pasamos al método `setLayout`.

En segundo lugar, cuando añadimos un componente a un contenedor con un `BorderLayout`, utilizamos un método `add` diferente que tiene un segundo parámetro. El valor del segundo parámetro es una de las constantes públicas `NORTH`, `SOUTH`, `EAST`, `WEST` y `CENTER`, que están definidas en la clase `BorderLayout`.

Código 11.5 Utilización de un `BorderLayout` para colocar los componentes.

```
contentPane = frame.getContentPane();

contentPane.setLayout(new BorderLayout());

filenameLabel = new JLabel();
contentPane.add(filenameLabel, BorderLayout.NORTH);

imagePanel = new ImagePanel();
contentPane.add(imagePanel, BorderLayout.CENTER);

statusLabel = new JLabel("Version 1.0");
contentPane.add(statusLabel, BorderLayout.SOUTH);
```

Ejercicio 11.21 Implemente y pruebe el código mostrado más arriba en su versión del proyecto.

Ejercicio 11.22 Experimente con otros gestores de diseño gráfico. Pruebe en su proyecto todos los gestores de diseño mencionados anteriormente y compruebe que se comportan como cabe esperar.

11.5.5 Filtros de imagen

Nos queda por hacer dos cosas antes de finalizar nuestra primera versión del visualizador de imágenes: añadir algunos filtros de imagen y añadir un menú *Help*. A continuación, vamos a agregar los filtros.

---

[5] Estrictamente hablando, la llamada a `setLayout` no sería necesaria aquí, ya que el gestor de diseño gráfico predeterminado del panel de contenido ya es un `BorderLayout`. Hemos incluido aquí la llamada por razones de claridad y legibilidad.

Los filtros de imagen constituyen el primer paso para manipular las imágenes. En último término, lo que deseamos es no solo poder abrir y visualizar imágenes, sino también poder manipularlas y volver a guardarlas en disco.

Aquí, vamos a empezar añadiendo tres filtros sencillos. Un filtro es una función que se aplica a la imagen completa. (Podría, por supuesto, modificarse para aplicarlo a una parte de una imagen, pero no vamos a hacer eso por el momento.)

Los tres filtros se denominan *darker* (oscurecer), *lighter* (aclarar) y *threshold* (umbral). *Darker* permite oscurecer la imagen completa, mientras que *lighter* permite aclararla. El filtro *threshold* pasa la imagen a escala de grises con solo unos pocos tonos de gris predefinidos. Esta técnica se denomina umbralización. Hemos elegido una umbralización de tres niveles, lo que quiere decir que emplearemos tres colores: negro, blanco y gris medio. Todos los píxeles que se encuentren en el tercio superior del rango de valores de brillo se considerarán blancos; todos los que se encuentren en el tercer inferior se transformarán en negros y los que se encuentren en el tercio intermedio serán grises.

Para conseguir esto tenemos que hacer dos cosas:

- Crear elementos de menú para cada filtro con un escucha de menú asociado.
- Implementar la propia operación del filtro.

Primero vayamos con los menús. En esto no nos encontramos con ninguna novedad: hay que añadir un poco más del mismo código de creación de menús que ya hemos escrito para el menú existente.

Tenemos que añadir las siguientes partes:

- Creamos un nuevo menú (clase `JMenu`) denominado *Filter* y lo añadimos a la barra de menús.
- Creamos tres elementos de menú (clase `JMenuItem`) denominados *Darker*, *Lighter* y *Threshold*, y los añadimos a nuestro menú *Filter*.
- A cada elemento de menú, le añadimos un escucha de acción, utilizando el código especial para las clases anónimas que hemos presentado al hablar de los otros elementos de menú. Los escuchas de acción deben invocar los métodos `makeDarker`, `makeLighter` y `threshold`, respectivamente.

Después de haber añadido los menús y creado los métodos (inicialmente vacíos) para gestionar las funciones de filtrado, necesitamos implementar cada filtro.

Los tipos más simples de filtro implican iterar a través de la imagen completa y realizar un cambio de algún tipo en el color de cada píxel. En el Código 11.6 se muestra un patrón para este proceso. Otros filtros más complicados podrían utilizar el valor de los píxeles adyacentes para ajustar el valor de cada píxel.

Ejercicio 11.23 Añada el nuevo menú y los elementos de menú a su versión del proyecto *image-viewer0-4*, como se describe aquí. Para añadir los escuchas de acción, necesitará crear los tres métodos `makeDarker`, `makeLighter` y `threshold` como métodos privados en su clase `ImageViewer`. Todos tienen un tipo de retorno `void` y no admiten parámetros. Estos métodos pueden disponer inicialmente de cuerpos vacíos o podrían limitase a imprimir que han sido invocados.

Código 11.6
Patrón para un proceso simple de filtrado.

```
int height = getHeight();
int width = getWidth();
for(int y = 0; y < height; y++) {
    for(int x = 0; x < width; x++) {
        Color pixel = getPixel(x, y);
        // alterar el valor de color del píxel;
        setPixel(x, y, pixel);
    }
}
```

La propia función de filtrado opera sobre la imagen, por lo que de acuerdo con las directrices del diseño dirigido por responsabilidad, debería implementarse en la clase `OFImage`. Por otro lado, la gestión de la invocación del menú también incluye algo del código relacionado con la GUI (por ejemplo, tenemos que comprobar si una imagen está abierta en el momento de invocar el filtro) y esto pertenece a la clase `ImageViewer`.

Como resultado de este razonamiento, creamos dos métodos, uno en `ImageViewer` y otro en `OFImage`, que compartirán el trabajo (Código 11.7 y Código 11.8). Podemos ver que el método `makeDarker` de `ImageViewer` contiene la parte de la tarea relacionada con la GUI (comprobar que tenemos cargada una imagen, mostrar un mensaje de estado, repintar el marco), mientras que el método `darker` de `OFImage` incluye el trabajo real de hacer que cada píxel de la imagen se oscurezca un poco.

Código 11.7
El método de filtrado en la clase `ImageViewer`.

```
public class ImageViewer
{
    // Se omiten campos, constructores y todos los restantes métodos.

    /**
     * Función 'Darker': oscurecer la imagen.
     */
    private void makeDarker()
    {
        if(currentImage != null) {
            currentImage.darker();
            frame.repaint();
            showStatus("Applied: darker");
        }
        else {
            showStatus("No image loaded.");
        }
    }
}
```

Código 11.8
Implementación de un filtro en la clase `OFImage`.

```
public class OFImage extends BufferedImage
{
    // Se omiten campos, constructores y todos los restantes métodos

    /**
     * Hacer que esta imagen sea un poco más oscura.
     */
    public void darker()
    {
        int height = getHeight();
        int width = getWidth();
        for(int y = 0; y < height; y++) {
            for(int x = 0; x < width; x++) {
                setPixel(x, y, getPixel(x, y).darker());
            }
        }
    }
}
```

Ejercicio 11.24 ¿Qué es lo que hace la llamada a método `frame.repaint()`, que puede ver en el método `makeDarker`?

Ejercicio 11.25 Podemos ver una llamada a un método `showStatus`, que es claramente una llamada a método interno. A partir del nombre, podemos adivinar que este método debería mostrar un mensaje de estado utilizando la etiqueta de estado que hemos creado anteriormente. Implemente este método en su versión del proyecto *imageviewer0-4*. (*Sugerencia*: examine el método `setText` de la clase `JLabel`.)

Ejercicio 11.26 ¿Qué sucede si se selecciona el elemento de menú *Darker* cuando no se ha abierto ninguna imagen?

Ejercicio 11.27 Explique detalladamente cómo funciona el método `darker` de `OFImage`. (*Sugerencia*: contiene otra llamada a método, a un método que también se denomina `darker`. ¿A qué clase pertenece este segundo método? Búsquelo.)

Ejercicio 11.28 Implemente el filtro *lighter* en `OFImage`.

Ejercicio 11.29 Implemente el filtro *threshold*. Para obtener el brillo de un píxel, puede consultar sus valores de rojo, verde y azul y sumarlos. La clase `Color` define referencias estáticas a objetos adecuados de color negro, blanco y gris.

Puede encontrar una implementación funcional de todo lo que hemos descrito hasta ahora en el proyecto *imageviewer1-0*. No obstante, debería intentar hacer estos ejercicios por sí mismo antes de mirar la solución.

11.5.6 Cuadros de diálogo

Nuestra última tarea para esta versión consiste en añadir un menú *Help* que tenga un elemento de menú denominado *About ImageViewer...* Al seleccionar este elemento de menú, aparecerá un cuadro de diálogo que mostrará un pequeño texto informativo.

Ahora tenemos que implementar el método `showAbout` para que muestre un cuadro de diálogo "About".

Ejercicio 11.30 De nuevo, añada un menú denominado *Help*. En él, añada un elemento de menú etiquetado como *About ImageViewer...*

Ejercicio 11.31 Añada un esqueleto de método (un método con un cuerpo vacío) denominado `showAbout`, y añada un escucha de acción al elemento de menú *About ImageViewer...* en el que se invoque a este método.

Una de las principales características de un cuadro de diálogo es si se trata de un cuadro de diálogo *modal* o no. Un cuadro de diálogo modal bloquea todas las interacciones con otras partes de la aplicación hasta que el cuadro de diálogo se haya cerrado. Obliga al usuario a tratar primero con lo que ese cuadro de diálogo le esté comunicando. Los cuadros de diálogo no modales permiten interactuar en otros marcos mientras están visibles.

Los cuadros de diálogo pueden implementarse de forma similar a nuestro `JFrame` principal. A menudo utilizan la clase `JDialog` para mostrar el marco.

Sin embargo, para los cuadros de diálogo modales con una estructura estándar, existen algunos métodos cómodos en la clase `JOptionPane` que hacen que resulte muy sencillo mostrar dichos cuadros de diálogo. `JOptionPane` tiene, entre otras cosas, métodos estáticos para mostrar tres tipos de cuadros de diálogo estándar. Estos tres tipos son:

- *Cuadro de diálogo de mensajes:* este cuadro de diálogo muestra un mensaje y tiene un botón *OK* para cerrar el cuadro.
- *Cuadro de diálogo de confirmación:* este cuadro de diálogo suele plantear una pregunta y tiene botones para que el usuario haga una selección; por ejemplo, *Yes*, *No* y *Cancel*.
- *Cuadro de diálogo de entrada:* este cuadro de diálogo incluye un indicativo y un campo de texto para que el usuario introduzca un texto.

Nuestro recuadro "About" es un cuadro de diálogo de mensaje simple. Buscando en la documentación de `JOptionPane`, encontramos que hay métodos estáticos, `showMessageDialog`, que permiten hacer esto.

Ejercicio 11.32 Localice la documentación de `showMessageDialog`. ¿Cuántos métodos hay con este nombre? ¿Cuáles son las diferencias entre ellos? ¿Cuál deberíamos usar para el recuadro "About"? ¿Por qué?

Ejercicio 11.33 Implemente el método `showAbout` en su clase `ImageViewer`, utilizando una llamada a un método `showMessageDialog`.

Ejercicio 11.34 Los métodos `showInputDialog` y `JOptionPane` permiten solicitar al usuario que introduzca una cierta entrada mediante un cuadro de diálogo, cuando sea necesario. Por otro lado, el componente `JTextField` permite mostrar un área permanente de texto dentro de una GUI. Localice la documentación para esta clase. ¿Qué entrada provoca que sea notificado un `ActionListener` asociado con un `JTextField`? ¿Se puede impedir a un usuario que edite el texto del campo? ¿Es posible que se notifique a un escucha la realización de cambios arbitrarios en el texto del campo? *Sugerencia*: ¿qué uso hace un `JTextField` de un objeto `Document`?

Puede ver un ejemplo de un `JTextField` en el proyecto *calculator* del Capítulo 7.

Después de estudiar la documentación, podemos implementar nuestro recuadro "About" haciendo una llamada al método `showMessageDialog`. La solución de código se muestra en el Código 11.9. Observe que hemos introducido una constante de cadena denominada `VERSION` para almacenar el número de versión actual.

Código 11.9
Visualización de un cuadro de diálogo modal.

```
private void showAbout()
{
    JOptionPane.showMessageDialog(frame,
                "ImageViewer\n" + VERSION,
                "About ImageViewer",
                JOptionPane.INFORMATION_MESSAGE);
}
```

Esta era la última tarea que nos quedaba por hacer para completar la "versión 1.0" de nuestra aplicación del visualizador de imágenes. Si ha hecho todos los ejercicios, debería disponer ahora de una versión del proyecto que puede abrir imágenes, aplicar filtros, mostrar mensajes de estado y mostrar un cuadro de diálogo.

El proyecto *imageviewer1-0* incluido en los proyectos del libro contiene una implementación de toda la funcionalidad que hemos presentado hasta el momento. Debería estudiar cuidadosamente este proyecto y compararlo con sus propias soluciones.

En este proyecto, hemos mejorado también el método `openFile` para incluir una mejor notificación de los errores. Si el usuario selecciona un archivo que no sea un archivo de imagen válido, debemos mostrar ahora un mensaje de error apropiado. Ahora que sabemos como diseñar cuadros de diálogo de mensajes, esto resulta fácil de hacer.

11.5.7 Resumen de la gestión del diseño gráfico

En esta sección, hemos añadido algunas clases personalizadas para tratar con las imágenes. Pero también (y lo que es más importante para nuestra GUI) hemos examinado la cuestión de la colocación de los componentes. Hemos visto cómo los contenedores y los gestores de diseño gráfico trabajan conjuntamente para conseguir la disposición en pantalla exacta que necesitamos.

Aprender a trabajar con gestores de diseño gráfico requiere una cierta experiencia y, a menudo, algo de experimentación a base de prueba y error. Sin embargo, con el tiempo, llegará a conocer bien los gestores de diseño gráfico.

Con esto hemos cubierto los fundamentos de todas las áreas importantes de programación de una GUI. En el resto del capítulo, podemos concentrarnos en la realización de los ajustes finales y en la implementación de una serie de mejoras de nuestra solución actual.

11.6 ImageViewer 2.0: mejora de la estructura del programa

La versión 1.0 de nuestra aplicación tiene una GUI utilizable y puede mostrar imágenes. También puede aplicar tres filtros básicos.

La siguiente idea obvia para mejorar nuestra aplicación consiste en añadir algunos filtros más interesantes. Antes de precipitarnos a hacerlo, pensemos en lo que esto implica.

Con la estructura actual de filtros, tenemos que hacer tres cosas para cada filtro:

1. Añadir un elemento de menú.
2. Añadir un método para gestionar la activación del menú en `ImageViewer`.
3. Añadir una implementación del filtro en `OFImage`.

Los puntos 1 y 3 son inevitables: necesitamos un elemento de menú y una implementación del filtro. Pero el punto 2 parece sospechoso: si examinamos esos métodos en la clase (en el Código 11.10 se muestran dos de ellos como ejemplo), parece que estuviéramos duplicando código. Estos métodos son esencialmente el mismo (salvo por algunos pequeños detalles), y para cada nuevo filtro tenemos que añadir otro de estos métodos.

Código 11.10
Dos de los métodos de gestión de filtros en `ImageViewer`.

```
private void makeLighter()
{
    if(currentImage != null) {
        currentImage.lighter();
        frame.repaint();
        showStatus("Applied: lighter");
    }
    else {
        showStatus("No image loaded.");
    }
}
private void threshold()
{
    if(currentImage != null) {
        currentImage.threshold();
        frame.repaint();
        showStatus("Applied: threshold");
    }
    else {
        showStatus("No image loaded.");
    }
}
```

Como sabemos, la duplicación de código es un indicio de mal diseño y debe evitarse. Vamos a solventar el problema refactorizando nuestro código.

En este caso, queremos encontrar un diseño que nos permita añadir nuevos filtros sin tener que agregar cada vez un nuevo método despachador para el filtro.

Para conseguir lo que deseamos, tenemos que evitar codificar cada nombre de método de filtrado (`lighter`, `threshold`, etc.) en nuestra clase `ImageViewer`. En lugar de ello, emplearemos una colección de filtros y escribiremos un único método de invocación de filtros que localice e invoque el filtro adecuado. Esto será similar en lo que respecta al estilo a la introducción de un método `act` al desacoplar el simulador de los tipos individuales de actor en el proyecto *foxes-and-rabbits* del Capítulo 10.

Para poder hacer esto, los propios filtros deberán transformarse en objetos, en lugar de ser solo nombres de métodos. Si queremos almacenarlos en una colección común, entonces todos los filtros necesitarán una superclase común, que denominaremos `Filter` y a la que dotaremos de un método `apply` (la Figura 11.11 muestra la estructura y el Código 11.11 el código fuente).

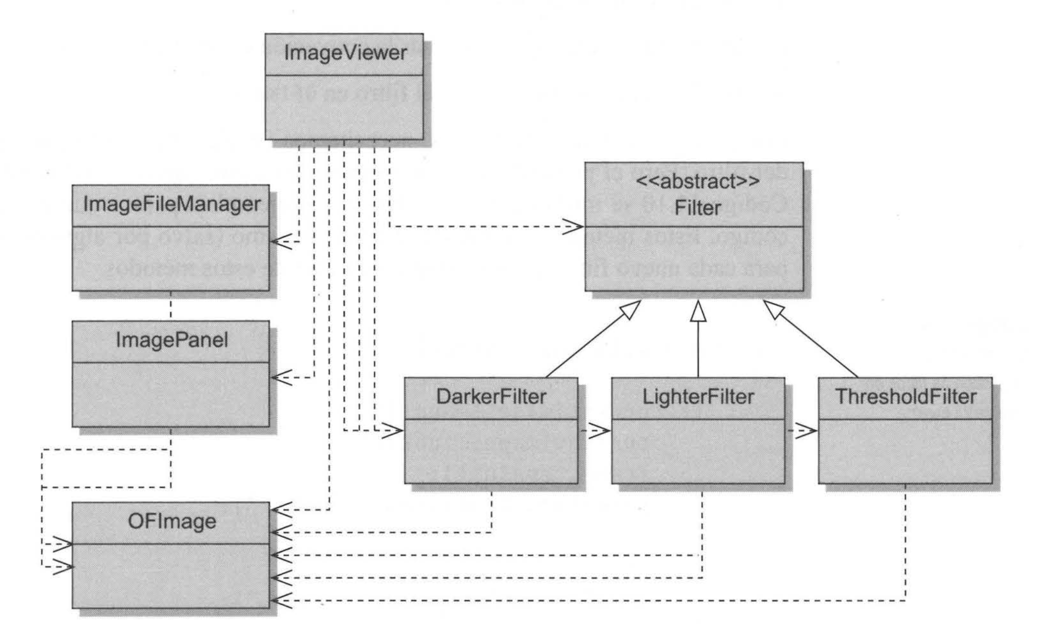


Figura 11.11 Estructura de clases considerando los filtros como objetos.

Cada filtro tendrá un nombre individual y su método `apply` aplicará ese tipo concreto de filtro a una imagen. Observe que se trata de una clase abstracta, ya que el método `apply` tiene que ser abstracto en este nivel, pero el método `getName` puede implementarse completamente, por lo que no es una interfaz.

Una vez escrita la superclase, no resulta difícil implementar los filtros específicos como subclases. Todo lo que tenemos que hacer es proporcionar una implementación del método `apply` que manipule una imagen (que se pasa como parámetro), utilizando los métodos `getPixel` y `setPixel`. El Código 11.12 muestra un ejemplo.

Como efecto colateral de esto, la clase `OFImage` pasa a ser mucho más simple, ya que pueden eliminarse de ella todos los métodos de filtrado. Ahora define únicamente los métodos `setPixel` y `getPixel`.

Código 11.11
Clase abstracta `Filter`: superclase para todos los filtros.

```
public abstract class Filter
{
    private String name;

    /**
     * Crear un nuevo filtro con un nombre especificado.
     */
    public Filter(String name)
    {
        this.name = name;
    }

    /**
     * Devolver el nombre de este filtro.
     *
     * @return el nombre de este filtro.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Aplicar este filtro a una imagen.
     *
     * @param image La imagen que hay que modificar
     *              mediante este filtro.
     */
    public abstract void apply(OFImage image);
}
```

Código 11.12
Implementación de una clase específica de filtro.

```
// Se omiten todos los comentarios.

public class DarkerFilter extends Filter
{
    public DarkerFilter(String name)
    {
        super(name);
    }

    public void apply(OFImage image)
    {
        int height = image.getHeight();
        int width = image.getWidth();
```

Código 11.12
(*continuación*)
Implementación de una clase específica de filtro.

```
          for(int y = 0; y < height; y++) {
              for(int x = 0; x < width; x++) {
                  image.setPixel(
                      x, y, image.getPixel(x, y).darker());
              }
          }
      }
  }
```

Una vez definidos nuestros filtros de esta forma, podemos crear objetos filtro y almacenarlos en una colección (Código 11.13).

Código 11.13
Adición de una colección de filtros.

```
public class ImageViewer
{
    // Se omiten otros campos y los comentarios.

    private List<Filter> filters;
    public ImageViewer()
    {
        filters = createFilters();
        ...
    }

    private List<Filter> createFilters()
    {
        List<Filter> filterList = new ArrayList<Filter>();
        filterList.add(new DarkerFilter("Darker"));
        filterList.add(new LighterFilter("Lighter"));
        filterList.add(new ThresholdFilter("Threshold"));

        return filterList;
    }

    // Se omiten otros métodos.
}
```

Ahora que disponemos de esta estructura, podemos hacer los dos cambios que necesitamos.

- Vamos a modificar el código que crea los elementos de menú de filtrado, para que itere a través de la colección de filtros. Para cada filtro, creará un elemento de menú y utilizará el método `getName` del filtro, con el fin de determinar la etiqueta del elemento.
- Habiendo hecho esto, podemos escribir un método `applyFilter` genérico que reciba un filtro como parámetro y aplique ese filtro a la imagen actual.

El proyecto *imageviewer2-0* incluye una implementación completa de estos cambios.

Ejercicio 11.35 Abra el proyecto *imageviewer2-0*. Estudie el código del nuevo método para crear y aplicar filtros de la clase `ImageViewer`. Preste una atención especial a los métodos `makeMenuBar` y `applyFilter`. Explique detalladamente cómo funciona la creación de los elementos del menú de filtrado y su activación. Dibuje un diagrama de objetos. Observe, en particular, que la variable `filter` en `makeMenuBar` ha sido declarada como `final`, como se ha explicado en la Sección 11.4.8. Asegúrese de comprender por qué esto es necesario.

Ejercicio 11.36 ¿Qué hará falta cambiar para añadir un nuevo filtro al visualizador de imágenes?

Ejercicio 11.37 *Ejercicio avanzado* Puede que haya observado que los métodos `apply` de todas las subclases de `Filter` tienen una estructura muy similar: iteran a través de la imagen completa y cambian el valor de cada píxel, independientemente de los píxeles adyacentes. Debería ser posible aislar esta duplicación de forma bastante similar a como hemos hecho a la hora de crear la clase `Filter`.

Cree un método en la clase `Filter` que itere a través de la imagen y aplique una transformación específica de cada filtro a cada píxel individual. Sustituya los cuerpos de los métodos `apply` en las tres subclases de `Filter` por una llamada a este método, pasando como parámetros la imagen y un objeto que pueda aplicar la transformación apropiada.

En esta sección, hemos realizado una refactorización pura. No hemos modificado la funcionalidad de la aplicación en absoluto, sino que hemos mejorado exclusivamente la estructura de la implementación para que los cambios futuros resulten más sencillos.

Ahora, después de finalizar la refactorización, debemos comprobar que toda la funcionalidad existente sigue siendo la esperada. En todos los proyectos de desarrollo, necesitamos fases como esta, porque no siempre se toman decisiones impecables de diseño desde el principio, y además las aplicaciones crecen y los requisitos cambian. Aun cuando nuestra tarea principal en este capítulo es la de trabajar con interfaces GUI, necesitábamos dar un paso atrás y refactorizar nuestro código antes de continuar. Este trabajo se verá recompensado a largo plazo, al hacer que todos los cambios ulteriores sean más sencillos.

En ocasiones, es tentador dejar las estructuras de las aplicaciones como están, aunque seamos conscientes de que no son adecuadas. Convivir con algo de duplicación de código puede ser más sencillo a corto plazo que llevar a cabo una cuidadosa refactorización. Es posible funcionar de esa manera durante un corto tiempo, pero en los proyectos que están pensados para sobrevivir durante un tiempo más largo, esto termina por generar problemas. Como regla general: ¡tómese su tiempo y mantenga limpio el código!

Ahora que hemos hecho esto, estamos listos para añadir algunos filtros adicionales.

Ejercicio 11.38 Añada un filtro *grayscale* a su proyecto. El filtro debe transformar la imagen en otra en blanco y negro con una serie de tonos de gris. Puede asignar a un píxel cualquier tono de gris dando a la tres componentes de color (rojo, verde, azul) el mismo valor. El brillo de cada píxel no debe modificarse.

Ejercicio 11.39 Añada un filtro *mirror* que actúe como un espejo, volteando la imagen horizontalmente. El píxel situado en la esquina superior izquierda debe moverse a la esquina superior derecha y viceversa, produciendo el efecto de visualizar la imagen en un espejo.

Ejercicio 11.40 Añada un filtro *invert* que invierta cada color. "Invertir" un color significa sustituir cada valor de color *x* por 255 – *x*.

Ejercicio 11.41 Añada un filtro *smooth* que "suavice" la imagen. Un filtro de suavizado sustituye cada valor de píxel por la media de sus píxeles adyacentes y de él mismo (nueve píxeles en total). Tiene que tener cuidado en los bordes de la imagen en los que no existen algunos de los píxeles adyacentes. También debe asegurarse de trabajar con una copia temporal de la imagen mientras la procesa, porque el resultado no será correcto si trabaja sobre la misma imagen. (¿A qué se debe esto?) Puede obtener fácilmente una copia de la imagen creando una nueva `OFImage` y pasando la imagen original como parámetro a su constructor.

Ejercicio 11.42 Añada un filtro *solarize*. La solarización es un efecto que puede obtenerse manualmente en los negativos fotográficos, volviendo a realizar una exposición de un negativo revelado. Podemos simular esto sustituyendo cada componente de color de cada píxel que tenga un valor *v* menor que 128 por 255 – *v*. Las componentes más brillantes (valor igual a 128 o mayor) no sufrirán modificación. (Este es un algoritmo muy simple de solarización –en la literatura técnica puede encontrar la descripción de algunos otros algoritmos más sofisticados.)

Ejercicio 11.43 Implemente un filtro *edge detection*, para la detección de bordes. Hágalo analizando los nueve píxeles de un cuadrado de tres por tres alrededor de cada píxel (de forma similar al filtro de suavizado) y luego configure el valor del píxel central para que sea igual a la diferencia entre los valores más alto y más bajo que haya encontrado. Haga esto para cada componente de color /rojo, verde, azul). El aspecto que se obtiene también es adecuado si invierte la imagen al mismo tiempo.

Ejercicio 11.44 Experimente con sus filtros aplicándolos a diferentes imágenes. Trate de aplicar varios filtros uno después de otro.

Una vez que haya implementado algunos filtros adicionales de su propia cosecha, debería cambiar el número de versión de su proyecto a "versión 2.1".

11.7 ImageViewer 3.0: más componentes de la interfaz

Antes de abandonar el proyecto de visualización de imágenes, queremos añadir unas últimas mejoras adicionales. En el proceso, echaremos un vistazo a dos componentes más de una GUI: botones y bordes.

11.7.1 Botones

Ahora queremos añadir al visualizador de imágenes la funcionalidad para cambiar el tamaño de la imagen. Haremos esto mediante dos funciones: *larger*, que duplica el tamaño de la imagen y *smaller*, que reduce su tamaño a la mitad. (Para ser exactos lo que duplicaremos o reduciremos a la mitad es la anchura y la altura, no el área.)

Una forma de hacer esto consiste en implementar filtros para estas tareas. Sin embargo, decidimos no hacerlo así. Hasta ahora, los filtros nunca han cambiado el tamaño de la imagen, y preferimos dejar que esto siga siendo así. En lugar de ello, vamos a introducir una barra de herramientas en el lado izquierdo de nuestro marco que contendrá dos botones etiquetados

como *Larger* y *Smaller* (Figura 11.12). Esto nos dará también la oportunidad de experimentar un poco con los botones, los contenedores y los gestores de diseño gráfico.

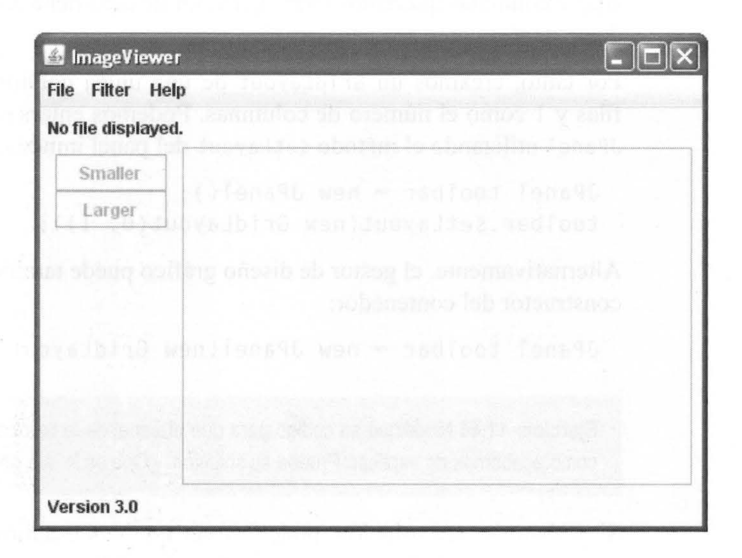


Figura 11.12 El visualizador de imágenes con botones de barra de herramientas.

Hasta ahora, nuestro marco utiliza un `BorderLayout`, en el que el área `WEST` está vacía. Podemos emplear esta área para añadir nuestros botones de la barra de herramientas. Sin embargo, hay un pequeño problema: el área `WEST` de un `BorderLayout` solo puede albergar un componente, mientras que nosotros tenemos dos botones.

La solución es simple: añadimos un `JPanel` al área `WEST` del marco (ya que sabemos que un `JPanel` es un contenedor) y luego colocamos los dos botones dentro del `JPanel`. El Código 11.14 muestra la solución que permite hacer esto.

Código 11.14 Adición de un panel de barra de herramientas con dos botones.

```
// Crear la barra de herramientas con los botones
JPanel toolbar = new JPanel();

smallerButton = new JButton("Smaller");
toolbar.add(smallerButton);

largerButton = new JButton("Larger");
toolbar.add(largerButton);

contentPane.add(toolbar, BorderLayout.WEST);
```

Ejercicio 11.45 Añada dos botones etiquetados como *Larger* y *Smaller* a su última versión del proyecto, utilizando un código similar al anterior. Pruebe su solución. ¿Qué es lo que observa?

Cuando probamos esta solución, vemos que funciona parcialmente, pero que no tiene el aspecto deseado. La razón es que `JPanel` utiliza, de manera predeterminada, un `FlowLayout`, y el `FlowLayout` dispone sus componentes horizontalmente. Lo que nos gustaría es colocarlos en vertical.

Podemos conseguir esto utilizando otro gestor de diseño gráfico. Uno que hace lo que queremos es `GridLayout`. Al crear un `GridLayout`, los parámetros del constructor determinan cuántas filas y columnas queremos tener. Un valor de cero tiene un significado especial aquí, queriendo decir "tantas como sea necesario".

Por tanto, creamos un `GridLayout` de una única columna utilizando 0 como el número de filas y 1 como el número de columnas. Podemos entonces usar este `GridLayout` para nuestro `JPanel` utilizando el método `setLayout` del panel inmeditamente después de crearlo:

```
JPanel toolbar = new JPanel();
toolbar.setLayout(new GridLayout(0, 1));
```

Alternativamente, el gestor de diseño gráfico puede también especificarse como parámetro del constructor del contenedor:

```
JPanel toolbar = new JPanel(new GridLayout(0, 1));
```

Ejercicio 11.46 Modifique su código para que el panel de la barra de herramientas utilice un `GridLayout`, como acabamos de explicar. Pruebe su solución. ¿Que es lo que observa?

Si probamos esta solución, podemos ver que nos estamos aproximando, pero que todavía no hemos conseguido lo que queremos. Nuestros botones son ahora mucho más grandes de lo que queríamos. La razón es que un contenedor en un `BorderLayout` (nuestra barra de herramientas `JPanel` en este caso) siempre cubre su área completa (el área `WEST` de nuestro marco). Y un `GridLayout` siempre redimensiona sus componentes para rellenar todo el contenedor.

Un `FlowLayout` no hace esto; por el contrario no tiene ningún problema en dejar algo de espacio vacío alrededor de los componentes. Por tanto, nuestra solución será emplear ambos: el `GridLayout` para colocar los botones en una columna y un `FlowLayout` a su alrededor para permitir algo de espacio vacío. Con eso terminamos teniendo un `GridLayout` dentro de un panel `FlowLayout`, dentro de un `BorderLayout`. El Código 11.15 muestra esta solución. Las construcciones como esta son muy comunes. A menudo tendremos que anidar varios contenedores dentro de otros contenedores, con el fin de conseguir el aspecto que deseamos.

Código 11.15
Utilización de un contenedor `GridLayout` anidado dentro de un contenedor `FlowLayout`.

```
// Crear la barra de herramientas con los botones.
JPanel toolbar = new JPanel();
toolbar.setLayout(new GridLayout(0, 1));

smallerButton = new JButton("Smaller");
toolbar.add(smallerButton);

largerButton = new JButton("Larger");
toolbar.add(largerButton);

// Añadir la barra de herramientas al panel,
// utilizando FlowLayout para el espaciado.
JPanel flow = new JPanel();
flow.add(toolbar);
contentPane.add(flow, BorderLayout.WEST);
```

Ahora nuestros botones tienen un aspecto bastante similar al que andamos buscando. Antes de añadir el toque final, podemos primero concentrarnos en hacer que los botones funcionen.

Tenemos que añadir dos métodos denominados, por ejemplo, `makeLarger` y `makeSmaller`, para llevar a cabo el trabajo real, y necesitamos añadir también escuchas de acción a los botones para invocar a esos métodos.

Ejercicio 11.47 En su proyecto, añada dos esqueletos de métodos denominados `makeLarger` y `makeSmaller`. Inicialmente, incluya simplemente una instrucción `println` dentro de los cuerpos de esos métodos, para ver cuándo han sido invocados. Los métodos pueden ser privados.

Ejercicio 11.48 Añada escuchas de acción a los dos botones que invoquen los dos nuevos métodos. Añadir escuchas de acción a los botones es idéntico a añadir escuchas de acción a los elementos de menú. Puede simplemente copiar el patrón de código del caso de los elementos de menú. Pruebe su solución. Asegúrese de que se invocan los métodos `makeSmaller` y `makeLarger` al activar los botones.

Ejercicio 11.49 Implemente adecuadamente los métodos `makeSmaller` y `makeLarger`. Para hacer esto, tendrá que crear una nueva `OFImage` con un tamaño diferente, copiar los píxeles de la imagen actual (mientras efectúa el cambio de escala para ampliar o reducir) y luego definir la nueva imagen como imagen actual. Al final de los métodos, debería invocar el método `pack` del marco para reordenar los componentes después del cambio de tamaño.

Ejercicio 11.50 Todos los componentes Swing tienen un método `setEnabled(boolean)` que puede activar y desactivar el componente. Los componentes desactivados suelen mostrarse en color gris claro y no reaccionan a la entrada del usuario. Cambie su visualizador de imágenes para que los dos botones de la barra de herramientas estén desactivados inicialmente. Se les deberá activar cuando se abra una imagen y volver a desactivar cuando se cierre.

11.7.2 Bordes

El último toque que queremos añadir a nuestra interfaz son algunos bordes internos. Los bordes pueden utilizarse para agrupar componentes o simplemente para añadir algo de espacio entre ellos. Todo componente Swing puede tener un borde.

Algunos gestores de diseño gráfico también aceptan parámetros en su constructor que definen su espaciado, y el gestor de diseño gráfico creará entonces el espaciado solicitado entre los componentes.

Los bordes más utilizados son `BevelBorder`, `CompoundBorder`, `EmptyBorder`, `EtchedBorder` y `TitledBorder`. Debería tratar de familiarizarse con ellos.

Vamos a hacer tres cosas para mejorar el aspecto de nuestra GUI:

- Añadir algo de espacio vacío alrededor de la parte exterior del marco.
- Añadir espacio entre los componentes del marco.
- Añadir una línea alrededor de la imagen.

El código para hacer esto se muestra en el Código 11.16. La llamada a `setBorder` para el panel de contenido con un `EmptyBorder` como parámetro añade espacio vacío alrededor de la parte

exterior del marco. Observe que ahora hacemos un cast del `contentPane` en un `JPanel`, ya que el supertipo `Container` no contiene el método `setBorder`.[6]

Código 11.16
Adición de espaciado con huecos y bordes.

```
JPanel contentPane = (JPanel) frame.getContentPane();
contentPane.setBorder(new EmptyBorder(12, 12, 12, 12));

// Especificar el gestor de diseño con un espaciado adecuado.
contentPane.setLayout(new BorderLayout(6, 6));

imagePanel = new ImagePanel();
imagePanel.setBorder(new EtchedBorder());
contentPane.add(imagePanel, BorderLayout.CENTER);
```

Crear el `BorderLayout` con dos parámetros `int` añade un espaciado entre los componentes que se encarga de colocar. Y finalmente, configurar un `EtchedBorder` para el `imagePanel` añade una línea con "relieve" alrededor de la imagen. (Los bordes está definidos en el paquete `javax.swing.border` –necesitamos añadir una instrucción de importación para este paquete.)

Todas las mejoras expuestas en esta sección se han implementado en la última versión de esta aplicación dentro de los proyectos del libro: *imageviewer3-0*. En esa versión, también hemos añadido una función *Save As* al menú *File* para poder guardar las imágenes en disco.

Y también hemos añadido un filtro más, denominado *Fish Eye*, que implementa una vista de ojo de pez para darle algunas ideas adicionales sobre las cosas que se pueden hacer. Pruébelo. Funciona especialmente bien con los retratos.

11.8 Ampliaciones adicionales

La programación de interfaces GUI con Swing es un área enormemente extensa. Swing dispone de muchos tipos distintos de componentes y de muchos contenedores y gestores de diseño gráfico distintos. Cada uno de ellos tiene muchos atributos y métodos.

Familiarizarse con la toda la librería Swing requiere su tiempo y no es algo que se pueda hacer en unas pocas semanas. Normalmente, a medida que vamos trabajando con interfaces GUI, lo que hacemos es continuar leyendo acerca de detalles que no conocíamos antes para terminar convirtiéndonos en expertos con el paso del tiempo.

El ejemplo expuesto en este capítulo, aunque contiene un montón de detalles, es solamente una breve introducción a la programación de interfaces GUI. Hemos explicado la mayor parte de los conceptos importantes, pero sigue habiendo una gran cantidad de funcionalidad que descubrir, la mayor parte de la cual cae fuera del alcance de este libro. Hay disponibles varias fuentes de información para ayudarle a continuar. Tendrá que consultar con frecuencia la documentación de la API para las clases Swing; es imposible trabajar sin ella. También hay disponibles muchos tutoriales GUI/Swing, tanto en forma de libro como en la Web.

---

[6] Utilizar un cast de esta manera solo funciona porque el tipo dinámico del panel de contenido ya es `JPanel`. Este cast no transforma el objeto panel de contenido en un `JPanel` en ningún sentido.

Un muy buen punto de partida para esto, como suele suceder a menudo, es el tutorial Java disponible públicamente en el sitio web de Oracle. Contiene una sección titulada *Creating a GUI with JFC/Swing* (`http://download.oracle.com/javase/tutorial/uiswing/index.html`).

En esta sección, hay muchas subsecciones interesantes. Una de las más útiles puede ser la sección titulada *Using Swing Components* y dentro de ella la subsección *How To...* Contiene las siguientes entradas: *How to Use Buttons, Check Boxes, and Radio Buttons*; *How to Use Labels*; *How to Make Dialogs*; *How to Use Panels*; etc.

De forma similar, la sección *Laying Out Components within a Container* también contiene una sección *How To...*,que informa acerca de todos los gestores de diseño gráfico (*layout manager*) disponibles.

Ejercicio 11.51 Localice la sección del Tutorial Java en línea denominada *Creating a GUI with JFC/Swing* (las secciones se denominan *trails* en el sitio web). Añádala a sus favoritos.

Ejercicio 11.52 ¿Qué es lo que ofrecen `CardLayout` y `GroupLayout` que sea distinto de los gestores de diseño que hemos visto en el capítulo?

Ejercicio 11.53 ¿Qué es un regulador (*slider*)? Localice una descripción y haga un resumen. Proporcione un corto ejemplo en código Java que permita crear y utilizar un regulador.

Ejercicio 11.54 ¿Qué es un panel con pestañas (*tabbed pane*)? Localice una descripción y haga un resumen. Proporcione ejemplos de para qué podría utilizarse un panel con pestañas.

Ejercicio 11.55 ¿Qué es un control rotatorio (*spinner*)? Localice una descripción y haga un resumen.

Ejercicio 11.56 Localice la aplicación de demostración *ProgressBarDemo*. Ejecútela en su computadora. Describa lo que hace.

Aquí es donde vamos a dejar el análisis del ejemplo del visualizador de imágenes. Sin embargo, los lectores interesados podrían ampliarlo en muchas direcciones. Utilizando la información del tutorial en línea, se pueden añadir numerosos componentes de interfaz.

Los siguientes ejercicios le proporcionarán algunas ideas y existen, obviamente, muchas más posibilidades.

Ejercicio 11.57 Implemente en su visualizador de imágenes una función *undo* (deshacer). Esta función deshace la última operación.

Ejercicio 11.58 Desactive todos los elementos de menú que no puedan utilizarse cuando no se esté visualizando una imagen.

Ejercicio 11.59 Implemente una función *reload* (recargar) que descarte todos los cambios realizados y la vuelva cargar del disco.

Ejercicio 11.60 La clase `JMenu` realmente es una subclase de `JMenuItem`. Esto significa que se pueden crear mneús anidados situando un `JMenu` dentro de otro. Añada un menú *Adjust* a la barra de menú. Anide

dentro de él un menú *Rotate* que permita girar la imagen 90 o 180 grados, en el sentido de las agujas del reloj o en el contrario. Implemente esta funcionalidad. El menú *Adjust* podría contener también elementos de menú que invoquen, por ejemplo, la funcionalidad *Larger* y *Smaller* existente.

Ejercicio 11.61 La aplicación redimensiona siempre su marco para garantizar que la imagen completa sea siempre visible. Pero disponer de un marco de gran tamaño no siempre es deseable. Lea la documentación de la clase `JScrollPane`. En lugar de añadir el `ImagePanel` directamente al panel de contenidos, coloque el panel en un `JScrollPane` y añada este panel desplazable al panel de contenido. Muestre una imagen grande y experimente redimensionando la ventana. ¿Qué diferencia implica la utilización de un panel desplazable? ¿Permite esto mostrar imágenes que de otra manera serían demasiado grandes para la pantalla?

Ejercicio 11.62 Modifique su aplicación para que pueda abrir varias imágenes al mismo tiempo (aunque solo se visualizará una imagen en cada momento). Luego añada un menú emergente (utilizando la clase `JComboBox`) para seleccionar la imagen que hay que visualizar.

Ejercicio 11.63 Como alternativa a la utilización de `JComboBox` como en el Ejercicio 11.62, utilice un panel con pestañas (clase `JTabbedPane`) para albergar múltiples imágenes abiertas.

Ejercicio 11.64 Implemente una función de presentación de diapositivas (*slide-show*) que permita seleccionar un directorio y luego muestre cada imagen de ese directorio durante un periodo de tiempo especificado (por ejemplo, 5 segundos).

Ejercicio 11.65 Una vez disponga de la presentación de diapositivas, añada un regulador (clase `JSlider`) que permita seleccionar una imagen de la presentación moviendo el regulador. Mientras se está mostrando la presentación de diapositivas, el regulador debería moverse para indicar el progreso.

11.9 Otro ejemplo: MusicPlayer

Hasta el momento en este capítulo hemos visto en detalle un ejemplo de una aplicación GUI. Ahora queremos presentar una segunda aplicación, con el fin de disponer de un segundo ejemplo del que extraer algunas lecciones. Este programa introduce algunos componentes GUI adicionales.

Este segundo ejemplo es una aplicación de un reproductor de música. No vamos a analizarla detalladamente, porque simplemente queremos usarla como base para que el lector estudie por su cuenta el código fuente, y también como fuente de fragmentos de código que el lector pueda copiar y modificar. Aquí, en este capítulo solo vamos a destacar unos cuantos aspectos seleccionados de la aplicación, aquellos en los que más merece la pena centrarse.

Ejercicio 11.66 Abra el proyecto *musicplayer*. Cree una instancia de `MusicPlayerGUI` y experimente con la aplicación.

El proyecto *musicplayer* proporciona una interfaz GUI a una serie de clases basadas en los proyectos *music-organizer* del Capítulo 4. Como allí, el programa localiza y reproduce

archivos MP3 almacenados en la subcarpeta *audio* de la carpeta del proyecto. Si dispone de sus propios archivos de sonido con el formato correcto, debería poder reproducirlos también copiándolos en la subcarpeta *audio* del proyecto.

El reproductor de música está implementado mediante tres clases: `MusicPlayerGUI`, `MusicPlayer` y `MusicFilePlayer`. Solo vamos a estudiar aquí la primera de ellas. `MusicFilePlayer` puede utilizarse esencialmente como una clase de librería; las instancias se crean junto con el nombre del archivo MP3 que hay que reproducir. Debe familiarizarse con su interfaz, pero no necesita comprender o modificar su implementación. (Aunque, por supuesto, puede si quiere estudiar esta clase también, aunque emplea conceptos que no vamos a explicar en este libro.)

A continuación incluimos algunas observaciones que merecen la pena acerca del proyecto *musicplayer*.

Separación modelo/vista

Esta aplicación utiliza una separación modelo/vista mejor que el ejemplo anterior. Esto significa que la funcionalidad de la aplicación (el modelo) está limpiamente separada de la interfaz de usuario (la GUI). Cada uno de los dos, el modelo y la vista, puede estar compuesto de varias clases, pero cada clase debe corresponder claramente a uno u otro grupo, con el fin de conseguir una separación clara. En nuestro ejemplo, la vista consta de una única clase GUI.

Separar la funcionalidad de la aplicación de la interfaz es un ejemplo de buena cohesión; hace que el programa sea más fácil de entender, de mantener y de adaptar a distintos requisitos (especialmente diferentes interfaces de usuario). Por ejemplo, sería bastante sencillo escribir una interfaz basada en texto para el reproductor de música, sustituyendo en la práctica la clase `MusicPlayerGUI`, pero sin efectuar modificaciones en la clase `MusicPlayer`.

Herencia de `JFrame`

En este ejemplo, estamos ilustrando la versión popular alternativa de creación de marcos que hemos mencionado al principio del capítulo. Nuestra clase GUI no instancia un objeto `JFrame`; en su lugar, amplía la clase `JFrame`. Como resultado, todos los métodos de `JFrame` que necesitamos invocar (como por ejemplo `getContentPane`, `setJMenuBar`, `pack`, `setVisible`, etc.) ahora pueden invocarse como métodos internos (heredados).

No hay ninguna razón de peso para preferir un estilo (utilización de una instancia de `JFrame`) y no el otro (heredar de `JFrame`). Básicamente, es una cuestión de preferencias personales, pero sí que tenemos que ser conscientes de que ambos estilos se utilizan ampliamente.

Visualización de imágenes estáticas

Es muy común querer mostrar una imagen en una GUI. La forma más fácil de hacer esto consiste en incluir en la interfaz un `JLabel` que tenga un gráfico como etiqueta (un `JLabel` puede mostrar texto, un gráfico o ambas cosas). El reproductor de sonido incluye un ejemplo de esto. El código fuente relevante es:

```
JLabel image = new JLabel(new ImageIcon("title.jpg"));
```

Esta instrucción cargará un archivo de imagen denominado *title.jpg* del directorio del proyecto, creará un icono con una imagen y luego creará un componente `JLabel` que muestre el icono. (El término "icono" parece sugerir que aquí estamos tratando únicamente con imágenes pequeñas pero la imagen puede ser, en la práctica, de cualquier tamaño.) Este método funciona para imágenes JPEG, GIF y PNG.

Cuadros combinados

El reproductor de sonido presenta un ejemplo de uso de un componente `JComboBox`. Un cuadro combinado (*combo box*) es un conjunto de valores, uno de los cuales está seleccionado en cada momento. El valor seleccionado se visualiza y la selección puede efectuarse a través de un menú desplegable. En el reproductor de sonido, el cuadro combinado se emplea para seleccionar una ordenación concreta de las pistas, por artista, título, etc.

Un `JComboBox` también puede ser editable, en cuyo caso los valores no todos está predefinidos, sino que pueden ser escritos por el usuario. En nuestro ejemplo, el componente no es editable.

Listas

El programa también incluye un ejemplo de una lista (clase `JList`) para la lista de pistas de audio. Una lista puede contener un número arbitrario de valores, pudiendo estar seleccionados uno o más de ellos. Los valores de la lista en este ejemplo son cadenas de caracteres, pero también son posibles otros tipos. Una lista no dispone automáticamente una barra de desplazamiento.

Barras de desplazamiento

Otro componente ilustrado en este ejemplo son las barras de desplazamiento.

Las barras de desplazamiento se pueden crear utilizando un contenedor especial, una instancia de la clase `JScrollPane`. En un panel de desplazamiento pueden colocarse objetos GUI de cualquier tipo y el panel de desplazamiento proporcionará las barras de desplazamiento necesarias si el objeto que alberga es demasiado grande como para visualizarlo en el espacio disponible.

En nuestro ejemplo, hemos colocado nuestra lista de pistas en un panel de desplazamiento. Después, el propio panel de desplazamiento se coloca en su contenedor padre. Las barras de desplazamiento solo son visibles cuando son necesarias. Puede comprobarlo añadiendo más pistas hasta que no quepan en el espacio disponible, o redimensionando la ventana para hacerla demasiado pequeña como para visualizar la lista actual.

Otros elementos ilustrados en este ejemplo son el uso de un regulador (que no tiene demasiada utilidad) y el uso del color para cambiar el aspecto de la aplicación. Cada uno de los elementos de la GUI dispone de muchos métodos para modificar el aspecto o el comportamiento –debería examinar la documentación de cualquier componente que le interese y experimentar modificando algunas propiedades de dicho componente.

Ejercicio 11.67 Modifique el reproductor de música para que muestre una imagen diferente en su parte central. Localice una imagen en la Web o diseñe la suya propia.

Ejercicio 11.68 Cambie los colores de los otros componentes (colores de primer plano y de fondo) para adaptarlos a la nueva imagen principal.

Ejercicio 11.69 Añada un nuevo componente para mostrar detalles de la pista actual cuando haya una reproduciéndose.

Ejercicio 11.70 Añada una funcionalidad *reload* al reproductor de música que lea de nuevo los archivos de la carpeta *audio*. Después, podrá copiar un nuevo archivo en la carpeta y cargarlo sin necesidad de salir del reproductor.

Ejercicio 11.71 Añada un elemento *Open* al menú *File*. Al activar este elemento de menú, debe presentar un cuadro de diálogo de selección de archivos que permita al usuario elegir el archivo de sonido que desea abrir. Si el usuario selecciona un directorio, el reproductor debe abrir todos los archivos de sonido contenidos en dicho directorio (como hace ahora con el directorio *audio*).

Ejercicio 11.72 Modifique el regulador para que el principio y el final (y posiblemente otras marcas intermedias) esté etiquetados con números. El principio debe ser cero y el final debe corresponderse con la longitud del archivo de música. La clase `MusicPlayer` dispone de un método `getLength` para obtener la longitud. Observe que el regulador actualmente no tiene ninguna funcionalidad.

Ejercicio 11.73 Modifique el reproductor de música para que al hacer doble clic sobre un elemento de la lista de pistas comience a reproducirse dicha pista.

Ejercicio 11.74 Mejore el aspecto de los botones. Todos los botones que no tengan ninguna función en algún momento deben mostrarse atenuados en color gris en ese instante y solo se deben activar cuando puedan ser utilizados de forma razonable.

Ejercicio 11.75 Actualmente, la visualización de las pistas es simplemente una lista `JList` de objetos `String`. Investigue si hay algún componente Swing disponible que proporcione una mayor sofisticación. Por ejemplo, ¿puede encontrar una forma de proporcionar una línea de cabecera y alinear el nombre del artista, el título y otras partes de la información de cada pista? Implemente esta solución en su versión.

Ejercicio 11.76 *Ejercicio avanzado* Haga que el regulador de posición se mueva a medida que se reproduce una pista.

11.10 Resumen

En este capítulo, hemos proporcionado una introducción a la programación de interfaces GUI utilizando las librerías Swing y AWT. Hemos expuesto las tres áreas conceptuales principales: la creación de componentes GUI, el diseño gráfico y el tratamiento de sucesos.

Hemos visto que la creación de una GUI suele comenzar con la creación de un marco de nivel superior, como por ejemplo un `JFrame`. El marco se rellena después con diversos componentes que proporcionan información y funcionalidad al usuario. Entre los componentes con los que nos hemos encontrado están los menús, elementos de menú, botones, etiquetas, bordes y otros.

Los componentes se colocan en pantalla con ayuda de contenedores y de gestores de diseño. Los contenedores albergan colecciones de componentes y cada contenedor dispone de un gestor de diseño que se encarga de disponer los componentes dentro del área de pantalla del contenedor. El anidamiento de contenedores, utilizando combinaciones de distintos gestores de diseño, es una forma muy común de conseguir la combinación deseada de tamaño y yuxtaposición de componentes.

Los componentes interactivos (los que pueden reaccionar a la entrada del usuario) generan sucesos al ser activados por un usuario. Otros objetos pueden convertirse en escuchas de sucesos y ser notificados de que dichos sucesos han tenido lugar mediante la implementación de interfaces estándar. Cuando el objeto escucha recibe una notificación, puede tomar las acciones adecuadas para tratar el suceso del usuario.

Hemos presentado el concepto de clases internas anónimas como técnica modular ampliable para la escritura de escuchas de sucesos.

Y por último hemos dado indicaciones relativas a una referencia y tutorial en línea que puede utilizarse para aprender más detalles no cubiertos en este capítulo.

Ejercicio 11.77 Añada una GUI al proyecto *world-of-zuul* del Capítulo 6. Cada sala debería tener asociada una imagen que se muestre cuando el jugador entre en la misma. Debería haber un área de texto no editable para mostrar la salida textual. Para introducir comandos, puede elegirse entre varias posibilidades: podemos dejar que la entrada siga estando basada en texto y utilizar un campo de texto (clase `JTextField`) para escribir los comandos o se pueden emplear botones para la entrada de comandos.

Ejercicio 11.78 Añada sonidos al juego *word-of-zuul*. Puede asociar sonidos individuales con las salas, los elementos o los personajes.

Ejercicio 11.79 Diseñe y construya una GUI para un editor de texto. Los usuarios deben poder introducir, editar, desplazar, etc. Considere las funciones para dar formato (tipos de letra, estilos y tamaños) y una función de recuento de caracteres/palabras. No hace falta implementar todavía las funciones de carga y guardado –puede esperar a implementarlas hasta haber leído el siguiente capítulo.

Términos introducidos en el capítulo

GUI, AWT, Swing, componente, diseño gráfico, suceso, tratamiento de sucesos, escucha de suceso, marco, barra de menús, menú, elemento de menú, panel de contenido, cuadro de diálogo modal, clase interna anónima, variable final

Resumen de conceptos

- **componentes** Una interfaz GUI se construye disponiendo componentes en pantalla. Los componentes se representan mediante objetos.
- **diseño gráfico** Para definir la colocación de los componentes de una GUI se utilizan gestores de diseño gráfico.

- **tratamiento de sucesos** El término tratamiento de sucesos hace referencia a la tarea de reaccionar a los sucesos provocados por el usuario, como los clics de ratón o la entrada a través del teclado.
- **formato de imagen** Las imágenes pueden almacenarse en diferentes formatos. Las diferencias afectan principalmente al tamaño del archivo y a la calidad de la imagen.
- **barra de menús, panel de contenido** Los componentes se colocan en un marco añadiéndolos a la barra de menús del marco o al panel de contenido.
- **escucha de sucesos** Un objeto puede escuchar los sucesos de los componentes implementando una interfaz de escucha de sucesos.
- **clases internas anónimas** Las clases internas anónimas son una estructura muy útil a la hora de implementar escuchas de sucesos.

CAPÍTULO

12 Tratamiento de errores

Principales conceptos explicados en el capítulo:

- programación defensiva
- informes de error
- generación y tratamiento de excepciones
- procesamiento básico de archivos

Estructuras Java explicadas en este capítulo:

`TreeMap, TreeSet, SortedMap, assert, exception, throw, throws, try, catch, File, FileReader, FileWriter, Path, Scanner, stream`

En el Capítulo 7, hemos visto que los errores lógicos en los programas son más difíciles de detectar que los errores sintácticos, porque el compilador no puede proporcionarnos ninguna ayuda en lo que respecta a los errores lógicos. Los errores lógicos surgen por diversas razones, que pueden solaparse en algunas situaciones:

- La solución a un problema se ha implementado de forma incorrecta. Por ejemplo, un problema relativo a la generación de ciertas estadísticas sobre determinados datos puede haber sida programado para calcular la media en lugar de la mediana (el valor "intermedio").
- Puede que se le pida a un objeto hacer algo que no es capaz de hacer. Por ejemplo, podría invocarse el método `get` de un objeto colección utilizando un valor de índice situado fuera del rango de valores válidos.
- Un objeto podría ser usado en formas que no hubieran sido anticipadas por el diseñador de la clase, y que eso conduzca a que el objeto quede en un estado incoherente o inapropiado. Esto sucede a menudo cuando se reutiliza una clase en un contexto distinto del original, quizá a través de los mecanismos de herencia.

Aunque la clase de estrategias de prueba expuestas en el Capítulo 7 puede ayudarnos a eliminar muchos errores lógicos antes de comenzar a utilizar nuestros programas, la experiencia sugiere que continuarán apareciendo fallos de programación. Además, hasta el programa más exhaustivamente probado puede fallar como resultado de circunstancias que están más allá del control del programador. Considere el caso de un explorador web al que se le pide que muestre una página web que no existe, o de un programa que trate de escribir datos en un disco en el que ya no queda más espacio. Esos problemas no son el resultado de errores lógicos de programación, pero podrían fácilmente hacer que un programa fallara si no se ha tenido en cuenta de antemano la posibilidad de que surjan esos problemas.

En este capítulo, vamos a ver cómo anticipar errores potenciales y cómo responder a situaciones de error a medida que vayan surgiendo durante la ejecución de un programa. Además, proporcionaremos algunas sugerencias acerca de cómo informar sobre los errores cuando estos se produzcan. También proporcionaremos una breve introducción al tema de cómo realizar la entrada/salida textual, ya que el procesamiento de archivos es una de esas situaciones en las que los errores pueden producirse con facilidad.

12.1 El proyecto *address-book*

Vamos a utilizar la familia de proyectos *address-book* para ilustrar algunos de los principios de tratamiento de errores y de generación de informes de errores que pueden surgir en muchas aplicaciones. Estos proyectos representan una aplicación que almacena detalles sobre contactos personales –nombre, dirección y número de teléfono– para un número arbitrario de personas. Una lista de contactos como esta podría ser empleada, por ejemplo, en un teléfono móvil o en un programa de correo electrónico. Los detalles de contacto están indexados en la libreta de direcciones tanto por nombre como por número de teléfono. Las clases principales que vamos a analizar son `AddressBook` (Código 12.1) y `ContactDetails`. Además, se proporciona la clase `AddressBookDemo` como una forma conveniente de definir una libreta de direcciones inicial con algunos datos de ejemplo.

Código 12.1
La clase `AddressBook`.

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.TreeSet;

/**
 * Una clase para mantener un número arbitrario de detalles
 * de contacto.
 * Los detalles están indexados tanto por nombre como por
 * número de teléfono.
 *
 * @author David J. Barnes y Michael Kölling.
 * @version 2011.07.31
 */
public class AddressBook
{
    // Almacenamiento para un número arbitrario de detalles.
    private TreeMap<String, ContactDetails> book;
    private int numberOfEntries;

    /**
     * Realizar la inicialización para la
     * libreta de direcciones.
     */
```

Código 12.1 (*continuación*) La clase `AddressBook`.

```
public AddressBook()
{
    book = new TreeMap<String, ContactDetails>();
    numberOfEntries = 0;
}

/**
 * Buscar un nombre o un número de teléfono y devolver
 * los detalles de contacto correspondientes.
 * @param key El nombre o número que hay que buscar.
 * @return Los detalles correspondientes a la clave.
 */
public ContactDetails getDetails(String key)
{
    return book.get(key);
}

/**
 * Devolver si está en uso o no la clave actual.
 * @param key El nombre o número que hay que buscar.
 * @return true si la clave está en uso,
 *         false en caso contrario.
 */
public boolean keyInUse(String key)
{
    return book.containsKey(key);
}

/**
 * Añadir un nuevo conjunto de detalles a la libreta
 * de direcciones.
 * @param details Los detalles que hay que asociar
 *                con la persona.
 */
public void addDetails(ContactDetails details)
{
    book.put(details.getName(), details);
    book.put(details.getPhone(), details);
    numberOfEntries++;
}

/**
 * Modificar los detalles previamente almacenados para
 * la clave indicada.
 * @param oldKey Una de las claves utilizadas para
 *               almacenar los detalles.
 * @param details Los detalles para la sustitución.
 */
```

Código 12.1
(*continuación*)
La clase
`AddressBook`.

```
    public void changeDetails(String oldKey,
                              ContactDetails details)
    {
        removeDetails(oldKey);
        addDetails(details);
    }
    /**
     * Buscar todos los detalles almacenados bajo una clave que
     * comienza con el prefijo indicado.
     * @param keyPrefix El prefijo de clave que hay que buscar.
     * @return Una matriz con los detalles que se hayan encontrado.
     */
    public ContactDetails[] search(String keyPrefix)
    {
        List<ContactDetails> matches =
                new LinkedList<ContactDetails>();
        // Encontrar claves que sean iguales o superiores
        // al prefijo.
        SortedMap<String, ContactDetails> tail =
                book.tailMap(keyPrefix);
        Iterator<String> it = tail.keySet().iterator();
        boolean endOfSearch = false;
        while(!endOfSearch && it.hasNext()) {
            String key = it.next();
            if(key.startsWith(keyPrefix)) {
                matches.add(book.get(key));
            }
            else {
                // Como la lista está ordenada, no vamos
                // a encontrar más.
                endOfSearch = true;
            }
        }
        ContactDetails[] results =
                new ContactDetails[matches.size()];
        matches.toArray(results);
        return results;
    }
    /**
     * Devolver el número de entradas que hay actualmente
     * en la libreta de direcciones.
     * @return El número de entradas.
     */
    public int getNumberOfEntries()
    {
        return numberOfEntries;
    }
```

Código 12.1 (*continuación*) La clase `AddressBook`.

```
    /**
     * Eliminar de la libreta de direcciones la entrada con
     * la clave indicada.
     * @param key Una de las claves de la entrada
     *            que hay que eliminar.
     */
    public void removeDetails(String key)
    {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
    /**
     * Devolver todos los detalles de contacto, ordenados
     * de acuerdo con el esquema de ordenación de la clase
     * ContactDetails.
     * @return Una lista ordenada de los detalles.
     */
    public String listDetails()
    {
        // Puesto que cada entrada está ordenada bajo dos claves,
        // es necesario construir un conjunto de los detalles
        // de contacto. Esto eliminará los duplicados.
        StringBuilder allEntries = new StringBuilder();
        Set<ContactDetails> sortedDetails =
                 new TreeSet<ContactDetails>(book.values());
        for(ContactDetails details : sortedDetails) {
            allEntries.append(details);
            allEntries.append('\n');
            allEntries.append('\n');
        }
        return allEntries.toString();
    }
}
```

Los nuevos detalles pueden almacenarse en la libreta de direcciones mediante su método `addDetails`. Esto presupone que los detalles representan un nuevo contacto, y no una modificación de los detalles correspondientes a un contacto ya existente. Para cubrir este último caso, el método `changeDetails` elimina una entrada antigua y la sustituye por los detalles revisados. La libreta de direcciones proporciona dos formas de extraer entradas: el método `getDetails` acepta como clave un nombre o un número de teléfono y devuelve los detalles correspondientes; el método `search` devuelve una matriz con todos los detalles que comiencen con una cadena de búsqueda especificada (por ejemplo, la cadena de búsqueda `"08459"` devolvería todas las entradas cuyos números telefónicos tengan ese prefijo).

Hay dos versiones introductorias del proyecto *address-book* para que las explore. Ambas proporcionan acceso a la misma versión de `AddressBook`, tal como se muestra en el Código

12.1. El proyecto *address-book-v1t* proporciona una interfaz de usuario basada en texto, con un estilo similar al de la interfaz del juego *zuul* presentado en el Capítulo 6. Hay disponibles actualmente una serie de comandos para enumerar el contenido de la libreta de direcciones, realizar búsquedas en esta y añadir una nueva entrada. Sin embargo, probablemente sea mucho más interesante la interfaz de la versión *address-book-v1g*, que incorpora una GUI simple. Experimente con ambas versiones para familiarizarse con lo que la aplicación puede hacer.

Ejercicio 12.1 Abra el proyecto *address-book-v1g* y cree un objeto `AddressBookDemo`. Invoque su método `showInterface` para visualizar la GUI e interactuar con la libreta de direcciones de ejemplo.

Ejercicio 12.2 Repita sus experimentos con la interfaz de texto del proyecto *address-book-v1t*.

Ejercicio 12.3 Examine la implementación de la clase `AddressBook` y evalúe si está bien escrita o no según su opinión. ¿Tiene alguna crítica específica que hacer?

Ejercicio 12.4 La clase `AddressBook` utiliza bastantes clases del paquete `java.util`; si no está familiarizado con alguno de ellas, consulte la documentación de la API para conseguir la información que le falte. ¿Cree que está justificado el uso de tantas clases de utilidad distintas? ¿Podría haberse utilizado un `HashMap` en lugar del `TreeMap`?

Si no está seguro, trate de cambiar el `TreeMap` por un `HashMap` y vea si el `HashMap` ofrece toda la funcionalidad requerida.

Ejercicio 12.5 Modifique las clases `CommandWords` y `AddressBookTextInterface` del proyecto *address-book-v1t* para proporcionar acceso interactivo a los métodos `getDetails` y `removeDetails` de `AddressBook`.

Ejercicio 12.6 La clase `AddressBook` define un campo para llevar la cuenta del número de entradas. ¿Cree que sería más apropiado calcular este valor directamente, a partir del número de entradas diferentes que componen el `TreeMap`? Por ejemplo, ¿podría pensar en alguna situación en la que el siguiente cálculo no produjera el mismo valor?

```
return book.size() / 2;
```

Ejercicio 12.7 ¿Cuán fácil cree que sería añadir un campo `String` para una dirección de correo electrónico a la clase `ContactDetails` y luego utilizar ese campo como tercera clave en `AddressBook`? No trate por el momento de implementar esta modificación.

12.2 Programación defensiva

12.2.1 Interacción cliente–servidor

Un `AddressBook` es un objeto servidor típico, que no inicia por su propia cuenta ningún tipo de acción; todas sus actividades están dirigidas por las solicitudes de los clientes. Los implementadores pueden adoptar al menos dos puntos de vista distintos a la hora de diseñar e implementar una clase servidora:

- Pueden asumir que los objetos cliente sabrán lo que están haciendo y que solo solicitarán los servicios de una forma coherente y bien definida.

- Pueden asumir que los objetos servidor operarán en un entorno esencialmente problemático, en el que hay que adoptar todas las posibles medidas para impedir que los objetos cliente utilicen el servidor de forma incorrecta.

Estos puntos de vista representan claramente dos extremos opuestos. En la práctica, el escenario más probable suele corresponderse con algún punto intermedio. La mayor parte de las interacciones de los clientes serán razonables, con algún intento ocasional de utilizar el servidor de manera incorrecta, bien como resultado de un error lógico de programación, o bien debido a que el programador del cliente no ha comprendido bien la forma en que se usa el servidor. Por supuesto, una tercera posibilidad es que haya un cliente intencionadamente hostil, que esté intentando hacer fallar el servidor o encontrar una debilidad en el mismo.

Estos distintos puntos de vista proporcionan una base útil a partir de la cual plantear preguntas como:

- ¿Cuántas comprobaciones de las solicitudes de los clientes deberían realizar los métodos del servidor?
- ¿Cómo debería informar el servidor a sus clientes acerca de los errores que se han producido?
- ¿Cómo puede un cliente anticipar el fallo de una solicitud enviada a un servidor?
- ¿Cómo debe un cliente tratar el fallo de una solicitud?

Si examinamos la clase `AddressBook` teniendo en cuenta estas cuestiones, veremos que la clase ha sido escrita en la confianza de que sus clientes siempre la van a utilizar de la manera apropiada. El Ejercicio 12.8 ilustra una de estas situaciones y cómo pueden surgir los problemas.

Ejercicio 12.8 Utilizando el proyecto *address-book-v1g*, cree un nuevo objeto `AddressBook` en el banco de objetos. Dicho objeto no contendrá ningún detalle de contacto. Llame ahora a su método `removeDetails` utilizando cualquier cadena de caracteres para la clave. ¿Qué sucede? ¿Comprende por qué sucede esto?

Ejercicio 12.9 Para un programador, la respuesta más fácil a la aparición de un error, consiste en permitir que el programa termine (es decir, que se produzca un "fallo catastrófico"). ¿Puede citar alguna situación en la que permitir simplemente que un programa termine podría ser terriblemente peligroso?

Ejercicio 12.10 Muchos programas comerciales contienen errores que no son tratados correctamente dentro del software y que hacen que se produzca un fallo catastrófico. ¿Cree que esto es inevitable? ¿Cree que es aceptable? Explique sus respuestas.

El problema con el método `removeDetails` es que asume que la clave que se le pasa es una clave válida de la libreta de direcciones. El método utiliza la supuesta clave para extraer los detalles de contacto asociados:

```
ContactDetails details = book.get(key);
```

Sin embargo, si el mapa no contiene esa clave concreta, entonces la variable `details` terminará almacenando el valor `null`. Esto, por sí mismo, no constituye un error; pero el error surge en la siguiente instrucción, en la que damos por supuesto que `details` hace referencia a un objeto válido:

```
book.remove(details.getName());
```

No está permitido invocar ningún método para una variable que contenga `null` y el resultado es un error en tiempo de ejecución. BlueJ informa de ese error indicando que se ha producido una excepción `NullPointerException` y resalta la instrucción que ha generado esa excepción. Más adelante en el capítulo, hablaremos de las excepciones detalladamente. Por ahora, digamos simplemente que si en una aplicación comercial se produjera un error como este, entonces la aplicación terminaría con un fallo catastrófico, es decir, terminaría de una manera no controlada, antes de haber completado su tarea.

Obviamente, aquí tenemos un problema, pero ¿de quién es la culpa? ¿Es culpa del objeto cliente por invocar el método con un valor de parámetro incorrecto? ¿Es culpa del objeto servidor por no tratar esta situación de manera apropiada? El desarrollador de la clase cliente podría decir que no hay nada en la documentación del método que diga que la clave tiene que ser válida. A la inversa, el desarrollador de la clase servidora podría argumentar que es obviamente erróneo tratar de borrar detalles de contacto utilizando una clave no válida. Nuestro objetivo en este capítulo no es resolver este tipo de disputas, sino tratar de impedir que surjan. Para ello, comenzaremos examinando la cuestión del tratamiento de errores desde el punto de vista de la clase servidora.

Ejercicio 12.11 Guarde con otro nombre una copia de los proyectos *address-book-v1* para poder trabajar con ella. Modifique el método `removeDetails` para evitar que surja una excepción `NullPointerException` si el valor de la clave no tiene ninguna entrada correspondiente dentro de la libreta de direcciones. Utilice una instrucción if. Si la clave no es válida, entonces el método no debería hacer nada.

Ejercicio 12.12 ¿Es necesario informar de la detección de una clave no válida en las llamadas a `removeDetails`? En caso informativo, ¿cómo podría informarse de ello?

Ejercicio 12.13 ¿Hay algún otro método en la clase `AddressBook` que sea vulnerable a errores similares? En caso afirmativo, trate de corregir esos métodos en su copia del proyecto. ¿Es aceptable en todos los casos que el método simplemente no haga nada cuando los valores de sus parámetros sean inapropiados? ¿Es necesario informar de los errores de alguna manera? En caso afirmativo, ¿cómo lo haría? ¿utilizaría siempre la misma forma para todos los posibles errores?

12.2.2 Comprobación de parámetros

Cuando más vulnerable es un objeto servidor es cuando su constructor y sus métodos reciben valores externos a través de sus parámetros. Los valores que se pasan a un constructor se utilizan para configurar el estado inicial del objeto, mientras que los valores que se pasan a un método se emplearán para influir sobre el efecto global de la llamada al método, y pueden cambiar el estado del objeto y el resultado que el método devuelve. Por tanto, es crucial que el objeto servidor sepa si puede confiar en la validez de los valores de los parámetros o si, por el contrario, necesita comprobar su validez por sí mismo. La situación actual tanto en la clase `ContactDetails` como en la clase `AddressBook` es que no hay ninguna comprobación de los valores de los parámetros. Como hemos visto en el caso del método `removeDetails`, esto puede llevar a que se produzca un error fatal en tiempo de ejecución.

Impedir una `NullPointerException` en `removeDetails` es relativamente sencillo, y el Código 12.2 ilustra cómo se puede hacer. Observe que, además de mejorar el código fuente del

método, hemos actualizado el comentario del método, con el fin de documentar el hecho de que las claves desconocidas serán ignoradas.

Código 12.2
Protección frente a una clave no válida en `removeDetails`.

```
/**
 * Eliminar de la libreta de direcciones la entrada con la
 * clave indicada. Si la clave no existe, no hacer nada.
 * @param key Una de las claves de la entrada que hay que eliminar.
 */
public void removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
}
```

Si examinamos todos los métodos de `AddressBook`, encontraremos que hay otros lugares en los que se podrían realizar unas mejoras similares:

- El método `addDetails` debería comprobar que su parámetro real no sea `null`.
- El método `changeDetails` debería comprobar tanto que la clave antigua existe, como que los nuevos detalles no son `null`.
- El método `search` debería comprobar que la clave no sea `null`.

Todos estos cambios se han implementado en las versiones de la aplicación que puede encontrar en los proyectos *address-book-v2g* y *address-book-v2t*.

Ejercicio 12.14 ¿Por qué cree que hemos considerado innecesario hacer cambios similares en los métodos `getDetails` y `keyInUse`?

Ejercicio 12.15 Al tratar con los errores de los parámetros, no hemos impreso ningún mensaje de error. ¿Cree que una instancia de `AddressBook` *debería* imprimir un mensaje de error cada vez que uno de sus métodos reciba un valor de parámetro incorrecto? ¿Hay alguna situación en la que un mensaje impreso de error sería inapropiado? Por ejemplo, ¿cree que son apropiados los mensajes de error mostrados en el terminal, si estamos utilizando la versión GUI del proyecto?

Ejercicio 12.16 ¿Cree que hay alguna otra comprobación que deberíamos realizar sobre los parámetros de los otros métodos para impedir que un objeto `AddressBook` funcione de manera incorrecta?

12.3 Generación de informes de error de servidor

Habiendo protegido un objeto servidor para evitar que realice una operación ilegal debido a valores incorrectos de los parámetros, podríamos pensar que eso es todo lo que tiene que hacer el desarrollador de un servidor. Sin embargo, idealmente deberíamos tratar de evitar

que esas situaciones de error pudieran siquiera llegar a plantearse. Además, a menudo sucede que los valores incorrectos de parámetros son el resultado de algún tipo de error de programación en el cliente que los ha suministrado. Por tanto, en lugar de limitarnos simplemente a programar para evitar el problema en el servidor, resulta conveniente que el servidor haga algún esfuerzo para indicar que se ha producido un problema, bien informando al propio cliente, bien informando a un usuario o al programador. De esta forma, existe la posibilidad de que llegue a corregirse un cliente que haya sido incorrectamente escrito. Pero observe que esas tres "audiencias" potenciales para la notificación serán a menudo completamente distintas.

¿Cuál es la mejor manera en que un servidor puede informar de los problemas cuando estos se producen? No hay una única respuesta a esta pregunta y la más apropiada dependerá a menudo del contexto en que se esté utilizando un objeto servidor concreto. En las siguientes secciones vamos a explorar una serie de opciones para los informes de error generados por un servidor.

Ejercicio 12.17 ¿Cuántas formas distintas se le ocurren de indicar que un método ha recibido valores de parámetros incorrectos o es incapaz, de alguna manera, de completar su tarea? Considere tantos tipos distintos de aplicaciones como pueda -por ejemplo: aplicaciones con una GUI; aplicaciones con una interfaz textual y un usuario humano; aplicaciones que no tienen usuarios interactivos, como el software incorporado en los sistemas de gestión del motor de los vehículos; el software en sistemas empotrados, como por ejemplo un cajero automático.

12.3.1 Notificación al usuario

La forma más obvia en la que un objeto puede responder cuando detecta que hay algún error es tratar de notificar de alguna manera al usuario de la aplicación. Las opciones principales son: imprimir un mensaje de error utilizando `System.out` o `System.err` o mostrar un mensaje de error en una ventana de alerta.

Los dos problemas principales con ambos enfoques son:

- Presuponen que la aplicación está siendo utilizada por un usuario humano que verá el mensaje de error. Hay muchas aplicaciones que se ejecutan de manera completamente independiente de cualquier usuario humano. Por ello, un mensaje de error o una ventana de error no llegarían a ser observados. De hecho, la computadora que está ejecutando la aplicación puede no tener conectado en absoluto ningún dispositivo de visualización.
- Aun cuando haya un usuario humano para ver el mensaje de error, será raro que ese usuario tenga la posibilidad de hacer algo para corregir el problema. ¡Imagine un usuario en un cajero automático al que se le presenta un mensaje informando de que se ha producido una `NullPointerException`! Solo en aquellos casos en los que la acción directa del usuario haya conducido a la aparición de problema –como por ejemplo, si ha suministrado datos no válidos a la aplicación– existirá la posibilidad de que el usuario pueda tomar algún tipo de acción correctora la siguiente vez o al menos evitar volver a cometer el mismo error.

Los programas que imprimen mensajes de error inútiles, en lugar de conseguir un efecto beneficioso, lo que hacen es desconcertar al usuario. Por tanto, excepto en un conjunto muy limitado de circunstancias, la notificación del error al usuario no es una solución general al problema de la generación de informes de error.

12.3.2 Notificación al objeto cliente

Un enfoque radicalmente distinto del que acabamos de exponer consiste en que el objeto servidor envíe una indicación al objeto cliente cuando algo salga mal. Hay dos formas principales de hacer esto:

- El servidor puede utilizar en sus métodos un tipo de retorno distinto de `void` para devolver un valor que indique si la llamada al método ha tenido éxito o ha fallado.
- Un servidor puede *generar una excepción* si algo sale mal. Esto nos permite introducir una nueva característica de Java, que también está presente en algunos otros lenguajes de programación. Describiremos esta característica detalladamente en la Sección 12.4.

Ambas técnicas tienen la ventaja de animar al programador del *cliente* a que tenga en cuenta que la llamada a un método puede fallar. Sin embargo, solo la decisión de generar una excepción impedirá de manera activa al programador del cliente que ignore las consecuencias del fallo de un método.

La primera de las dos técnicas es fácil de introducir en un método que normalmente tendría un tipo de retorno `void`, como por ejemplo `removeDetails`. Si se sustituye el tipo `void` por un tipo `boolean`, entonces el método puede devolver `true` para indicar que la eliminación ha tenido éxito y `false` para indicar que ha fallado por alguna razón (Código 12.3).

Código 12.3
Un tipo de retorno `boolean` para indicar el éxito o el fallo de un método.

```
/**
 * Eliminar de la libreta de direcciones la entrada con la clave
 * indicada. La clave debe ser una que esté actualmente en uso.
 * @param key Una de las claves de la entrada que hay que eliminar.
 * @return true si la entrada se ha eliminado con éxito,
 *         false en caso contrario.
 */
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

Esto permite a un cliente utilizar una instrucción if para proteger las instrucciones que dependan de la adecuada eliminación de una entrada:

```
if(contacts.removeDetails("...")) {
    // Entrada eliminada con éxito. Continuar de la forma normal.
    ...
}
```

```
else {
    // La eliminación ha fallado.
    // Intentar una recuperación si es posible.
    ...
}
```

Cuando un método de servidor ya tiene un tipo de retorno distinto de `void` –que impide en la práctica devolver un valor de diagnóstico de tipo `boolean`–, puede seguir habiendo una manera de utilizar el tipo de retorno para indicar que se ha producido un error. Esto será así si alguno de los valores del rango correspondiente al tipo de retorno está disponible para actuar como un valor de diagnóstico de error. Por ejemplo, el método `getDetails` devuelve un objeto `ContactDetails` correspondiente a una clave especificada, y el ejemplo siguiente asume que una clave concreta permitirá localizar un conjunto válido de detalles de contacto:

```
// Enviar a David un mensaje de texto.
ContactDetails details = contacts.getDetails("David");
String phone = details.getPhone();
...
```

Una forma en la que el método `getDetails` puede indicar que la clave es no válida o no está siendo utilizada es hacer que devuelva un valor `null` en lugar de un objeto `ContactDetails` (Código 12.4).

Código 12.4
Devolución de un valor de diagnóstico de error, utilizando un valor fuera del rango permitido.

```
/**
 * Buscar un nombre o un número de teléfono y devolver los
 * detalles de contacto correspondientes.
 * @param key El nombre o número que hay que buscar.
 * @return Los detalles correspondientes a la clave o
 *         null si la clave no está en uso.
 */
public ContactDetails getDetails(String key)
{
    if(keyInUse(key)) {
        return book.get(key);
    }
    else {
        return null;
    }
}
```

Esto permitiría a un cliente examinar el resultado de la llamada y luego continuar con el flujo normal de control o tratar de recuperarse del error:

```
ContactDetails details = contacts.getDetails("David");
if(details != null) {
    // Enviar un mensaje de texto a David.
    String phone = details.getPhone();
    ...
}
```

```
else {
    // No se ha podido encontrar la entrada.
    // Intentar una recuperación si es posible.
    ...
}
```

Resulta común que los métodos que devuelven referencias a objetos utilicen el valor `null` como indicación de fallo o de error. Con los métodos que devuelven valores de tipo primitivo, en ocasiones habrá un *valor fuera de límites* que puede cumplir un papel similar: por ejemplo, el método `indexOf` de la clase `String` devuelve un valor negativo para indicar que no ha conseguido encontrar el carácter buscado.

Ejercicio 12.18 ¿Cree que los diferentes estilos de interfaz de las versiones *v2t* y *v2g* del proyecto *address-book* implican que debería haber una diferencia en la manera en que se informa de los errores a los usuarios?

Ejercicio 12.19 Utilizando una copia del proyecto *address-book-v2t*, modifique la clase `AddressBook` para proporcionar información acerca de un fallo a un cliente, cuando un método haya recibido valores de parámetro incorrectos o no pueda, por alguna otra razón, completar su tarea.

Ejercicio 12.20 ¿Cree que una llamada al método `search` que no encuentre ninguna correspondencia requiere una notificación de error? Justifique su respuesta.

Ejercicio 12.21 ¿Qué combinaciones de valores de parámetros serían inadecuadas para pasárselas al constructor de la clase `ContactDetails`?

Ejercicio 12.22 ¿Tiene un constructor alguna forma de indicar a un cliente que no puede configurar correctamente el estado de un nuevo objeto? ¿Qué debería hacer un constructor si recibe valores de parámetro inapropiados?

Claramente, no puede utilizarse un valor fuera de límites cuando todos los valores del rango correspondiente al tipo de retorno tengan ya un significado válido para el cliente. En tales casos, será necesario recurrir a la técnica alternativa de *generación de una excepción* (véase la Sección 12.4), que de hecho ofrece algunas ventajas significativas. Para entender por qué es así, merece la pena analizar dos problemas asociados con el uso de valores de retorno como indicadores de fallo o de error:

- Lamentablemente, no hay ninguna forma de obligar al cliente a comprobar las propiedades de diagnóstico del valor de retorno. En consecuencia, un cliente podría perfectamente continuar como si nada hubiera sucedido y entonces terminaría con una `NullPointerException`; o peor que eso: podría incluso utilizar el valor de retorno de diagnóstico como si fuera una valor de retorno normal, dando lugar a un error lógico muy difícil de diagnosticar.
- En algunos casos, podríamos estar utilizando el valor de diagnóstico para dos propósitos completamente distintos. Este sería el caso en los métodos `removeDetails` (Código 12.3) y `getDetails` (Código 12.4) modificados. Uno de esos propósitos es comunicar al cliente de si la solicitud ha tenido éxito o no. El otro consiste en indicarle que hay algo erróneo en su solicitud, como por ejemplo que ha pasado un valor de parámetro incorrecto.

En muchos casos, una solicitud *sin éxito* no representará un error lógico de programación, mientras que una solicitud *incorrecta* casi siempre lo representa. Como se trata de dos

situaciones muy distintas, lo que cabría esperar es que las posteriores acciones del cliente fueran también diferentes. Lamentablemente, no hay ninguna forma generalmente satisfactoria de resolver este conflicto utilizando simplemente valores de retorno.

12.4 Principios de la generación de excepciones

Generar una excepción es la forma más efectiva de la que dispone un objeto servidor para indicar que no puede terminar una llamada a uno de sus métodos. Una de las principales ventajas que tiene esto con respecto a utilizar un valor de retorno especial es que es (casi) imposible que un cliente ignore el hecho de que se ha generado una excepción y continúe su procesamiento como si nada hubiera pasado. Si el cliente no trata la excepción, la aplicación terminará inmediatamente.[1] Además, el mecanismo de excepciones es independiente del valor de retorno de un método, así que puede utilizarse para todos los métodos, sin importar su tipo de retorno.

Un punto importante que hay que recordar a lo largo de las explicaciones que siguen es que, cuando hablamos de excepciones, el lugar en el que se descubre un error será diferente de aquel en el se intenta recuperarse de ese error (si es que llega a efectuarse tal intento). El error será descubierto dentro del método del servidor, mientras que la recuperación se hará en el cliente. Si fuera posible efectuar la recuperación en el lugar donde se descubre el error, entonces no tendría ningún sentido generar una excepción.

12.4.1 Generación de una excepción

El Código 12.5 muestra cómo se genera una excepción utilizando una *instrucción throw*. Aquí, el método `getDetails` genera una excepción para indicar que no tiene sentido pasar un valor `null` para la clave porque no es una clave válida.

Código 12.5
Generación de una excepción.

```
/**
 * Buscar un nombre o número de teléfono y devolver los
 * detalles de contacto correspondientes.
 * @param key El nombre o número que hay que buscar.
 * @return Los detalles correspondientes a la clave,
 *         o null si no hay ninguna que se corresponda.
 * @throws IllegalArgumentException si la clave no es válida.
 */
public ContactDetails getDetails(String key)
{
    if(key == null){
        throw new IllegalArgumentException(
            "null key in getDetails");
    }
    return book.get(key);
}
```

[1] Esto es exactamente lo que ya hemos experimentado cuando nuestros programas morían de repente debido a una `NullPointerException` o una `IndexOutOfBoundsException`.

Concepto:

Una **excepción** es un objeto que representa los detalles de un fallo del programa. La excepción se genera para indicar que se ha producido un fallo.

Hay dos etapas en el proceso de generación de una excepción. Primero se crea un objeto excepción utilizando la palabra clave `new` (en este caso, un objeto `IllegalArgumentException`); después, se envía el objeto utilizando la palabra clave `throw`. Estas dos etapas suelen casi siempre combinarse en una única instrucción:

```
throw new TipoExcepcion("cadena-opcional-diagnóstico");
```

Cuando se crea un objeto excepción, puede pasarse una cadena de diagnóstico a su constructor. Esta cadena estará posteriormente a disposición del receptor de la excepción, a través de los métodos `getMessage` y `toString` del objeto excepción. Si esa excepción no se trata, la cadena se muestra también al usuario y esto conduce a la terminación de programa. El tipo de excepción que hemos utilizado aquí, `IllegalArgumentException`, está definido en el paquete `java.lang` y se suele utilizar normalmente para indicar que se ha pasado un valor de parámetro real incorrecto a un método o a un constructor.

El Código 12.5 también ilustra que la documentación `javadoc` de un método puede ampliarse para incluir detalles de las excepciones que genere, utilizando el marcador `@throws`.

12.4.2 Excepciones comprobadas y no comprobadas

Un objeto excepción es siempre una instancia de una clase que pertenece a una jerarquía de herencia especial. Podemos crear nuevos tipos de excepción creando subclases de esta jerarquía (Figura 12.1). Estrictamente hablando, las clases de excepción son siempre subclases de la clase `Throwable` definida en el paquete `java.lang`. Nosotros vamos a seguir el convenio de definir y utilizar clases de excepción que sean subclases de la clase `Exception`, también definida en `java.lang`.[2] El paquete `java.lang` define una serie de clases de excepción comunes, con las que puede que ya se haya encontrado inadvertidamente a la hora de desarrollar programas, como por ejemplo `NullPointerException`, `IndexOutOfBoundsException` y `ClassCastException`.

Java divide las clases de excepción en dos categorías: *excepciones comprobadas* y *excepciones no comprobadas*. Todas las subclases de la clase estándar Java `RuntimeException` son excepciones no comprobadas; todas las clases restantes de `Exception` son excepciones comprobadas.

Simplificando ligeramente, la diferencia es la siguiente: las excepciones comprobadas están pensadas para aquellos casos en los que el cliente debería esperar que una operación pueda fallar (por ejemplo, si trata de escribir en disco debería anticipar que el disco puede estar lleno). En tales casos, el cliente estará obligado a comprobar si la operación ha tenido éxito. Las excepciones no comprobadas están pensadas para aquellos casos que nunca deberían fallar durante la operación normal –usualmente indican un error del programa. Por ejemplo, ningún programador trataría conscientemente de extraer un elemento de una posición de una lista que no exista, por lo que si lo hace, se generará una excepción no comprobada.

Lamentablemente, saber qué categoría de excepción generar en cualquier circunstancia concreta, no es una ciencia exacta, aunque podemos ofrecerle los siguientes consejos de carácter general:

[2] `Exception` es una de las dos subclases directas de `Throwable`; la otra es `Error`. Las subclases de `Error` suelen estar reservadas para los errores del sistema de tiempo de ejecución, más que para errores sobre los que el programador disponga de control.

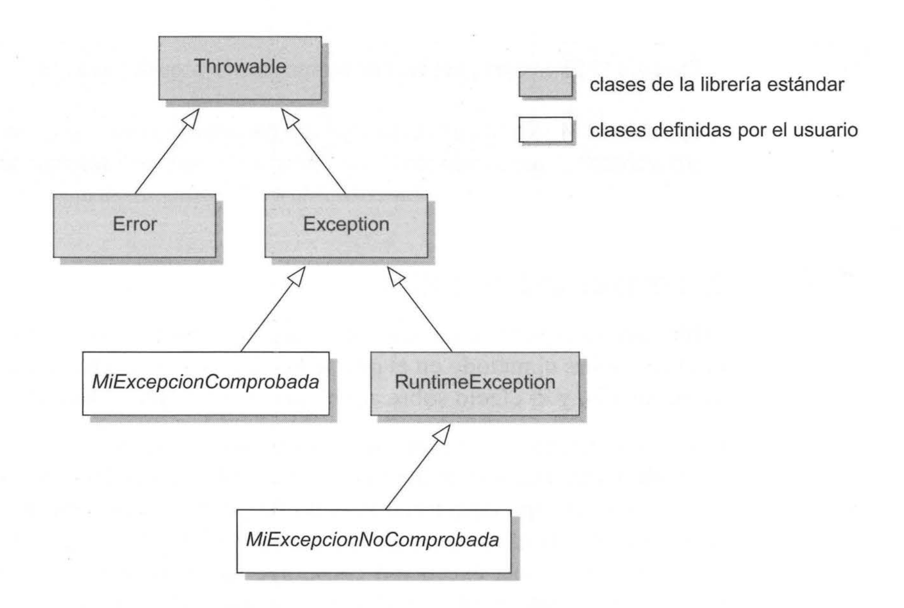


Figura 12.1
La jerarquía de clases de excepción.

- Una regla práctica consiste en utilizar excepciones no comprobadas para aquellas situaciones que deberían a conducir a un fallo del programa –normalmente, porque se sospecha que existe algún error lógico dentro del programa que le impedirá continuar haciendo su tarea. De aquí se sigue que las excepciones comprobadas deberían emplearse cuando exista la posibilidad de que el cliente lleve a cabo una recuperación. Un problema con este modo de actuar es que presupone que el servidor es lo suficientemente consciente del contexto en el que está siendo utilizado, como para ser capaz de determinar si es posible o imposible que el cliente pueda llevar a cabo una recuperación.
- Otra regla práctica consiste en emplear excepciones no comprobadas para aquellas situaciones que pudieran razonablemente ser evitadas. Por ejemplo, invocar un método para una variable que contiene `null` es el resultado de un error de programación lógico completamente evitable, y el hecho de que `NullPointerException` sea una excepción no comprobada encaja con este modelo. De aquí se deduce que las excepciones comprobadas deberían utilizarse para aquellas situaciones de fallo que caen fuera del control del programador, como por ejemplo que el disco se llene al tratar de escribir en un archivo o que una operación de red falle porque se ha interrumpido una conexión de red inalámbrica.

En Java, las reglas formales que gobiernan el uso de las excepciones son significativamente distintas para las excepciones comprobadas y las no comprobadas, y expondremos detalladamente las diferencias en las Secciones 12.4.4 y 12.5.1, respectivamente. En términos simples, esas reglas garantizan que un objeto cliente que invoque un método que pueda generar una excepción comprobada siempre contendrá código que prevea la posibilidad de que surja un problema y que intente tratar el problema en caso de que este se produzca.[3]

---

[3] De hecho, resulta muy fácil que el desarrollador del cliente respete las reglas, en principio, pero luego no intente recuperarse apropiadamente del problema, lo que pervierte el objetivo del mecanismo de excepciones.

Ejercicio 12.23 Enumere tres tipos de excepciones del paquete `java.io`.

Ejercicio 12.24 ¿Es `SecurityException` del paquete `java.lang` una excepción comprobada o no comprobada? ¿Y qué sucede con `NoSuchMethodException`? Justifique sus respuestas.

12.4.3 El efecto de una excepción

¿Qué sucede cuando se genera una excepción? Son dos los efectos que hay que considerar: el efecto sobre el método en el que se ha descubierto el problema y en el que se ha generado la excepción, y el efecto sobre aquel que ha invocado el método problemático.

Cuando se genera una excepción, la ejecución del método actual finaliza inmediatamente; no continúa hasta alcanzar el final del cuerpo del método. Una consecuencia de esto es que un método con un tipo de retorno distinto de `void` no está obligado a ejecutar una instrucción de retorno en aquellos casos en los que se genere una excepción. Esto es razonable, ya que la generación de una excepción constituye una indicación de la incapacidad de ese método para continuar con su ejecución normal, lo que incluye no ser capaz de devolver un resultado válido. Podemos ilustrar este principio con la siguiente versión alternativa del cuerpo del método mostrado en el Código 12.5:

```
if(key == null) {
    throw new IllegalArgumentException("null key in getDetails");
}
else {
    return book.get(key);
}
```

La ausencia de una instrucción de retorno en la ruta de ejecución que termina generando una excepción es perfectamente aceptable. De hecho, el compilador indicará un error si se escribe cualquier instrucción después de una instrucción throw, porque esas instrucciones nunca podrían llegar a ser ejecutadas.

El efecto que una excepción tiene sobre aquel punto del programa en el que se ha invocado al método problemático es algo más complejo. En particular, el efecto completo dependerá de si se ha escrito o no algún código para *capturar* la excepción. Considere la siguiente llamada errónea a `getDetails`:

```
AddressDetails details = contacts.getDetails(null);
// La siguiente instrucción nunca llegará a ser ejecutada.
String phone = details.getPhone();
```

Podemos decir que, en todos los casos, la ejecución de estas instrucciones tendrá un carácter incompleto; la excepción generada por `getDetails` interrumpirá la ejecución de la primera instrucción, por lo que no se realizará ninguna asignación a la variable `details`. Como resultado, la segunda instrucción tampoco se ejecutará.

Esto ilustra claramente el poder que las excepciones tienen a la hora de impedir que un cliente pueda continuar como si nada hubiera pasado independientemente del hecho de que ha surgido un problema. Lo que suceda en la práctica a continuación dependerá de si la excepción ha sido capturada o no. Si no se la captura, entonces el programa simplemente terminará, con una

indicación de que se ha generado una excepción `IllegalArgumentException` no capturada. Explicaremos cómo capturar una excepción en la Sección 12.5.2.

12.4.4 Utilización de excepciones no comprobadas

Concepto:

Una **excepción no comprobada** es un tipo de excepción cuyo uso no requiere ninguna comprobación por parte del compilador.

Las excepciones no comprobadas son las más fáciles de utilizar desde el punto de vista de un programador, porque el compilador impone muy pocas reglas en lo que a su utilización respecta. Esto es precisamente lo que significa "no comprobadas": el compilador no aplica ninguna comprobación especial ni el método en el que se genera una excepción no comprobada ni en el lugar desde el que dicho método ha sido invocado. Una clase de excepción será no comprobada si es una subclase de la clase `RuntimeException`, definida en el paquete `java.lang`. Todos los ejemplos que hemos utilizado hasta ahora para ilustrar el mecanismo de generación de excepciones se referían a excepciones no comprobadas. Así que poco tenemos que añadir aquí acerca de cómo generar una excepción no comprobada –basta con emplear una instrucción throw.

Si también seguimos el convenio de que deben emplearse excepciones no comprobadas en aquellas situaciones en las que el resultado esperado sea la terminación del programa –es decir, cuando la excepción no vaya a ser capturada–, entonces tampoco hace falta explicar nada más acerca de lo que tiene que hacer aquel que invocó el método, porque se limitará a no hacer nada y hacer que el programa falle. Sin embargo, si existe la necesidad de capturar una excepción no comprobada, entonces *puede* escribirse una rutina de tratamiento de excepción adecuada, igualmente que se hace con las excepciones comprobadas. En la Sección 12.5.2 se explica cómo hacer esto.

Ya hemos visto un ejemplo de uso de la excepción no comprobada `IllegalArgumentException`. Esa excepción es generada por un constructor o método para indicar que los valores de sus parámetros son inadecuados. Por ejemplo, el método `getDetails` también podría decidir generar esta excepción si la cadena que se pasa como clave está en blanco (Código 12.6).

Código 12.6
Comprobación de un valor de parámetro ilegal.

```
/**
 * Buscar un nombre o número de teléfono y devolver los
 * detalles de contacto correspondientes.
 * @param key El nombre o número que hay que buscar.
 * @throws IllegalArgumentException si la clave no es válida.
 * @return Los detalles correspondientes a la clave,
 *         o null si no hay ninguna que se corresponda.
 */
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new IllegalArgumentException("null key in getDe-
tails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                "Empty key passed to getDetails");
    }
    return book.get(key);
}
```

Merece la pena que los métodos realicen una serie de comprobaciones de validez sobre sus parámetros antes de continuar con el propósito principal del método. Esto hace que sea menos probable que el método tenga que llevar a cabo parcialmente sus acciones antes de generar una excepción debido a la presencia de valores de parámetros incorrectos. Una razón concreta para evitar ese tipo de situaciones es que una modificación parcial de un objeto tiene grandes posibilidades de dejarlo en un estado incoherente de cara a un uso futuro. Si un método falla por cualquier razón, lo ideal es que se deje el objeto para el que fue llamado en el mismo estado en el que se encontraba antes de intentar llevar a cabo la operación.

Ejercicio 12.25 Repase todos los métodos de la clase `AddressBook` y decida si hay alguna situación más en la que esos métodos debieran generar una excepción `IllegalArgumentException`. Añada las necesarias comprobaciones y las instrucciones de generación de excepciones.

Ejercicio 12.26 Si no lo ha hecho todavía, añada documentación javadoc para describir las excepciones generadas por los métodos de la clase `AddressBook`.

Ejercicio 12.27 `UnsupportedOperationException` es una excepción no comprobada definida en el paquete `java.lang`. ¿Cómo podría utilizarse en una implementación de la interfaz `java.util.Iterator` para impedir la eliminación de elementos de una colección mientras se está iterando a través de ella? Pruebe esta solución en la clase `LogfileReader` del proyecto *weblog-analyzer* del Capítulo 4.

12.4.5 Cómo impedir la creación de objetos

Un uso importante de las excepciones consiste en impedir que se creen objetos si estos no pueden colocarse en un estado inicial válido. Este tipo de situaciones serán normalmente consecuencia de que se pasen a un constructor valores de parámetro inapropiados. Podemos ilustrar esto mediante la clase `ContactDetails`. El constructor es actualmente bastante benévolo con los valores de parámetro que recibe: no rechaza los valores `null`, sino que los sustituye por cadenas de caracteres vacías. Sin embargo, en la libreta de direcciones necesita al menos un nombre o un número de teléfono para cada entrada con el fin de utilizarlos como valor unívoco de índice, por lo que será imposible indexar una entrada en la que tanto el campo `name` como el campo `phone` estén en blanco. Podemos reflejar este requisito impidiendo la construcción de un objeto `ContactDetails` que no disponga de detalles válidos para la clave. El proceso de generación de una excepción en un constructor es exactamente el mismo que cuando se genera una excepción dentro de un método. El Código 12.7 muestra el constructor revisado que impedirá que una entrada tenga simultáneamente en blanco los campos `name` y `phone`.

Una excepción generada en un constructor tendrá el mismo efecto sobre el cliente que una excepción generada en un método. Por tanto, el siguiente intento de crear un objeto `ContactDetails` no válido fracasará completamente; *no dará como resultado* que se almacene un valor `null` en la variable:

```
ContactDetails badDetails = new ContactDetails("", "", "");
```

Código 12.7
El constructor de la clase `ContactDetails`.

```
/**
 * Configurar los detalles de contacto. Todos los detalles se
 * recortan, para eliminar los espacios en blanco finales.
```

Código 12.7
(continuación)
El constructor de la clase `ContactDetails`.

```
 * El nombre o el número de teléfono no deben estar en blanco.
 * @param name El nombre.
 * @param phone El número de teléfono.
 * @param address La dirección.
 * @throws IllegalStateException si tanto el nombre como el número
 *                              de teléfono están en blanco.
 */
public ContactDetails(String name, String phone, String address)
{
    // Usar cadena en blanco si alguno de los parámetros es null.
    if(name == null) {
        name = "";
    }
    if(phone == null) {
        phone = "";
    }
    if(address == null) {
        address = "";
    }

    this.name = name.trim();
    this.phone = phone.trim();
    this.address = address.trim();

    if(this.name.length() == 0 && this.phone.length() == 0) {
        throw new IllegalStateException(
                  "Either the name or phone must not be blank.");
    }
}
```

12.5 Tratamiento de excepciones

Los principios de la generación de excepciones se aplican por igual tanto a las excepciones comprobadas como a las no comprobadas, pero las reglas de Java implican que el tratamiento de excepciones solo es obligatorio con las excepciones comprobadas. Una clase de excepción comprobada es aquella que sea subclase de `Exception` pero no de `RuntimeException`. Cuando se utilizan excepciones comprobadas, hay que seguir varias reglas más, porque el compilador impone una serie de comprobaciones tanto en el método que genera una excepción comprobada, como en todos aquellos que invoquen a dicho método.

12.5.1 Excepciones comprobadas: la cláusula throws

El primer requisito del compilador es que un método que genere una excepción comprobada debe declarar que lo hace así mediante una *cláusula throws* añadida a la cabecera del método.

Concepto:

Una **excepción comprobada** es un tipo de excepción cuyo uso requiere comprobaciones adicionales por parte del compilador. En particular, las excepciones comprobadas en Java requieren que se usen cláusulas *throws* e instrucciones *try*.

Un método que genere una `IOException` del paquete `java.io` podría tener la siguiente cabecera:[4]

```
public void saveToFile(String destinationFile)
    throws IOException
```

Está permitido utilizar una cláusula `throws` para las excepciones no comprobadas, pero el compilador no lo exige. Recomendamos que se utilice solo la cláusula `throws` para enumerar las excepciones comprobadas generadas por un método.

Es importante distinguir entre la cláusula `throws` añadida a la cabecera de un método y el comentario javadoc `@throws` que precede al método. Este último es completamente opcional para ambos tipos de excepciones. De todos modos, recomendamos que se incluya la documentación javadoc tanto para la excepciones comprobadas como para las no comprobadas. De esta manera, cualquiera que desee utilizar un método que genere una excepción tendrá disponible el máximo de información posible.

12.5.2 Anticipando las excepciones: la instrucción try

El segundo requisito, al utilizar excepciones comprobadas, es que el llamante de un método que genere una excepción comprobada debe tomar medidas para tratar con esa excepción. Esto implica habitualmente escribir una *rutina de tratamiento de excepciones* en la forma de una *instrucción try*. Las instrucciones try prácticas tiene el formato general mostrado en el Código 12.8. En ese fragmento se presentan dos nuevas palabras clave Java: `try` y `catch`, que marcan un *bloque try* y un *bloque catch*, respectivamente.

Código 12.8
Los bloques *try* y *catch* de una rutina de tratamiento de excepciones.

```
try {
    Proteger aquí una o más instrucciones.
}
catch(Exception e) {
    Informar aquí de la excepción y recuperarse de la misma.
}
```

Concepto:

El código de programa que protege las instrucciones en las que podría generarse una excepción se denomina **rutina de tratamiento de excepción**. Proporciona código para informar de la excepción y/o recuperarse de la misma, en caso de que se genere una.

Suponga que tenemos un método que guarda el contenido de una libreta de direcciones en un archivo. Le pedimos al usuario de alguna forma que proporcione el nombre de un archivo (quizá a través de una ventana de la GUI con un cuadro de diálogo) y después invocamos el método `saveToFile` de la libreta de direcciones para escribir la lista en el archivo. Si no tuviéramos que tener en cuenta las excepciones, entonces escribiríamos esto de la siguiente forma:

```
String filename = solicitar-un-archivo-al-usuario;
addressbook.saveToFile(filename);
```

Sin embargo, como el proceso de escritura podría terminar con una excepción, es preciso colocar la llamada a `saveToFile` dentro de un bloque try para demostrar que hemos tenido eso en cuenta. El Código 12.9 ilustra cómo tenderíamos a escribir esto anticipándonos a un posible fallo.

[4] Observe que la palabra clave utilizada aquí es `throws` y no `throw`.

Código 12.9
Una rutina de tratamiento de excepciones.

```
String filename = null;
try {
    filename = solicitar-un-archivo-al-usuario;
    addressbook.saveToFile(filename);
    successful = true;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    successful = false;
}
```

En un bloque try puede incluirse cualquier número de excepciones, por lo que de hecho tenderemos a colocar allí no solo la instrucción que puede fallar sino también todas aquellas otras instrucciones que estén relacionadas con ella de alguna manera. La idea es que un bloque try representa una secuencia de acciones que queremos tratar como una sola unidad lógica, reconociendo que puede fallar en algún punto.[5] El bloque catch intentará entonces tratar con la situación que se ha producido o informar acerca del problema, si es que se genera alguna excepción como consecuencia de la ejecución de cualquiera de las instrucciones contenidas dentro del bloque try asociado. Observe que, debido a que los bloques try y catch hacen uso de la variable `filename`, esa variable tendrá que ser declarada fuera de la instrucción try en este ejemplo, por razones de ámbitos de las variables.

Para comprender cómo funciona una rutina de tratamiento de excepciones, es esencial comprender que *una excepción impide que continúe en el llamante el flujo normal de control.* Una excepción interrumpe la ejecución de las instrucciones del llamante, por lo que cualquier instrucción situada inmediatamente después de la instrucción problemática no se ejecutará. Eso hace que se plantee la cuestión: "¿En qué punto se reanuda la ejecución en el llamante?". Una instrucción try proporciona la respuesta: si se genera una excepción debido a una instrucción invocada dentro del bloque try, entonces la ejecución se reanuda en el correspondiente bloque catch. Por tanto, si consideramos el ejemplo del Código 12.9, el efecto de que se genere una excepción `IOException` como resultado de la llamada a `saveToFile` será que el control se transferirá desde el bloque try al bloque catch, como se muestra en el Código 12.10.

Código 12.10
Transferencia de control en una instrucción try.

1. Excepción generada aquí **2.** El control se transfiere aquí.

```
try {
    filename = request-a-file-from-the-user;
    addressbook.saveToFile(filename);
    successful = true;
}
catch(IOException e) {
    System.out.println("Unable to save to " + filename);
    successful = false;
}
```

[5] Véase en el Ejercicio 12.30 un ejemplo de lo que puede suceder si una instrucción que podría dar como resultado una excepción es tratada de forma aislada con respecto a las instrucciones que la rodean.

Las instrucciones contenidas en un bloque try se conocen con el nombre de *instrucciones protegidas*. Si no surge ninguna excepción durante la ejecución de las instrucciones protegidas, entonces se saltará al bloque catch después de alcanzarse el final del bloque try. La ejecución continuará con lo que quiera que haya después de la instrucción try/catch completa.

Un bloque catch indica el tipo de excepción para el que está diseñado, encerrándolo entre paréntesis directamente después de la palabra `catch`. Además del nombre del tipo de excepción, también incluye un nombre de variable (que tradicionalmente es `e` o `ex` simplemente), que puede utilizarse para hacer referencia al objeto excepción generado. Disponer de una referencia a este objeto puede ser útil a la hora de proporcionar información que permita ayudar a recuperarse del problema, o bien a la hora de informar de que ese problema se ha producido –por ejemplo, accediendo a cualquier mensaje de diagnóstico asociado al objeto excepción por el método que ha generado la excepción. Una vez completado el bloque catch, el control *no* vuelve a la instrucción que ha provocado la excepción.

Ejercicio 12.28 El proyecto *address-book-v3t* incluye la generación de algunas excepciones no comprobadas cuando hay valores de parámetro iguales a `null`. El código fuente del proyecto incluye también la clase de excepción comprobada `NoMatchingDetailsException`, que actualmente no se utiliza. Modifique el método `removeDetails` de `AddressBook` para que genere esta excepción si su parámetro de clave no se corresponde con ninguna clave que esté siendo actualmente utilizada. Añada una rutina de tratamiento de excepciones al método `remove` de `AddressBookTextInterface` para capturar esta excepción e informar de que se ha producido.

Ejercicio 12.29 Utilice la excepción `NoMatchingDetailsException` en el método `changeDetails` de `AddressBook`. Mejore la interfaz de usuario de modo que puedan modificarse los detalles de una entrada existente. Capture las excepciones en `AddressBookTextInterface` que surjan del uso de una clave que no se corresponda con ninguna de las entradas existentes, e informe de que se han producido esas excepciones.

Ejercicio 12.30 ¿Por qué la siguiente no sería una forma adecuada de utilizar una rutina de tratamiento de excepciones? ¿Cree que este código se podría compilar y ejecutar?

```
Person p = null;
try {
    // La búsqueda podría fallar.
    p = database.lookup(details);
}
catch(Exception e) {
}
System.out.println("The details belong to: " + p);
```

Observe que en todos los ejemplos de instrucciones try que hemos visto, la excepción no es generada *directamente* por las instrucciones del bloque try que está en el objeto cliente. En lugar de ello, la excepción surge *indirectamente*, al ser pasada desde un método del objeto servidor, el cual ha sido invocado desde las instrucciones contenidas en el bloque try. Esta es la estructura usualmente utilizada, y encerrar una instrucción throw directamente dentro de una instrucción try será casi seguro un error.

12.5.3 Generación y captura de múltiples excepciones

En ocasiones, un método genera más de un tipo de excepción, para poder indicar que se han producido diferentes tipos de problemas. Cuando se trata de excepciones comprobadas, es necesario enumerarlas todas en la cláusula throws del método separadas por comas. Por ejemplo: :

```
public void process()
    throws EOFException, FileNotFoundException
```

Una rutina de tratamiento de excepciones debe gestionar todas las excepciones comprobadas generadas a partir de sus instrucciones protegidas, por lo que una instrucción try puede contener varios bloques catch, como se muestra en el Código 12.11. Observe que puede utilizarse el mismo nombre de variable para el objeto excepción en todos los casos.

Código 12.11
Varios bloques catch en una instrucción try.

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException e) {
    // Llevar a cabo una acción apropiada para la excepción de
    // fin de archivo.
    ...
}
catch(FileNotFoundException e) {
    // Llevar a cabo una acción apropiada para la excepción de
    // archivo no encontrado.
    ...
}
```

Cuando una llamada a método situada dentro de un bloque try genera una excepción, se comprueban los bloques catch en el orden en que aparecen escritos, hasta que se encuentra una correspondencia con el tipo de excepción. Por tanto, si se genera una `EOFException`, entonces el control se transferirá al primer bloque catch, y si se genera una `FileNotFoundException`, el control se transferirá al segundo bloque catch. Una vez alcanzado el final de uno de los bloques catch, la ejecución continúa después del último de los bloques catch.

Si se desea, puede emplearse el polimorfismo para evitar tener que escribir múltiples bloques catch. Sin embargo, esto podría tener como contrapartida no ser capaces de llevar a cabo acciones de recuperación específicas para cada tipo de excepción. En el Código 12.12, el único bloque catch que se muestra se encargará de tratar *todas* las excepciones generadas por las instrucciones protegidas. Esto se debe a que el proceso de búsqueda de correspondencias para las excepciones que se encarga de localizar un bloque catch apropiado simplemente se limita a comprobar que el objeto excepción sea una instancia del tipo indicado en el bloque. Como todas las excepciones son subtipos de la clase `Exception`, el mismo bloque capturará todas las excepciones, ya sean comprobadas o no comprobadas. Teniendo en cuenta la naturaleza del proceso de búsqueda de correspondencia para las excepciones, se deduce que el orden de los bloques catch dentro de una misma instrucción try es importante, y que ningún

bloque catch para un determinado tipo de excepción puede estar situado después del bloque correspondiente a uno de sus supertipos (porque el bloque correspondiente al supertipo, situado antes que el otro, siempre hará que se produzca una correspondencia, antes de comprobar el bloque correspondiente al subtipo). El compilador nos dará un error si tratamos de hacer esto.

Código 12.12
Captura de todas las excepciones mediante un único bloque catch.

```
try {
    ...
    ref.process();
    ...
}
catch(Exception e) {
    // Llevar a cabo una acción apropiada para todas
    // las excepciones.
    ...
}
```

12.5.4 Multicaptura en Java 7

En Java 7 se ha introducido una nueva característica que permite gestionar múltiples excepciones en un único bloque catch. Esto reduce la duplicación de código resultante cuando se necesite llevar a cabo la misma acción de recuperación para diferentes tipos de excepción. Los diferentes tipos de excepción se escriben delante del nombre de la variable de excepción, separados por el símbolo "|". Consulte el Código 12.13.

Código 12.13
Multicaptura en Java 7.

```
try {
    ...
    ref.process();
    ...
}
catch(EOFException | FileNotFoundException e) {
    // Llevar a cabo una acción apropiada para ambas excepciones.
    ...
}
```

Java 7 también introdujo otra nueva característica en la instrucción try, denominada *try con recursos* (try-with-resources) o ARM (*Automatic Resource Management*, Gestión automática de recursos). Hablaremos de esto más adelante en la Sección 12.9, junto con el tema de la gestión de archivos.

Ejercicio 12.31 Mejore las instrucciones try que ha escrito como soluciones a los Ejercicios 12.28 y 12.29, para que puedan tratar excepciones comprobadas y no comprobadas en diferentes bloques catch.

Ejercicio 12.32 ¿Qué hay de erróneo en la siguiente instrucción try?

```
Person p = null;
try {
    p = database.lookup(details);
    System.out.println("The details belong to: " + p);
}
catch(Exception e) {
    // Tratar todas las excepciones comprobadas...
    ...
}
catch(RuntimeException e) {
    // Tratar todas las excepciones no comprobadas ...
    ...
}
```

12.5.5 Propagación de una excepción

Hasta el momento, hemos sugerido que las excepciones deben capturarse y tratarse en la primera oportunidad disponible. Es decir, una excepción generada en un método denominado `process` debería ser capturada y tratada en el método que haya llamado a `process`. De hecho, esto no se hace siempre así, ya que Java permite que una excepción sea *propagada* desde el método cliente hasta el llamante del método cliente y posiblemente más allá. Para propagar una excepción, un método simplemente se limita a no incluir una rutina de tratamiento de excepciones que proteja la instrucción que puede generar esa excepción. Sin embargo, para las excepciones comprobadas, el compilador requiere que el método que está propagando la excepción incluya una cláusula throws, aun cuando no vaya a crear y enviar él mismo la excepción. Esto significa que con frecuencia vamos a encontrarnos con métodos que disponen de una cláusula throws, pero que no incluyen ninguna instrucción throw dentro del cuerpo del método. La propagación se utiliza de manera común cuando el método que hace la invocación no puede o no necesita llevar a cabo ninguna acción de recuperación por sí mismo, pero esas acciones de recuperación sí pueden ser posibles o necesarias en otras llamadas de mayor nivel. También es común la propagación en los constructores, cuando fallan las acciones tomadas por un constructor para configurar un nuevo objeto y el propio constructor es incapaz de recuperarse.

Si la excepción que se está propagando es del tipo no comprobado, entonces la cláusula throws es opcional, y nosotros preferimos omitirla.

12.5.6 La cláusula finally

Una instrucción try puede incluir un tercer componente que es opcional. Se trata de la *cláusula finally* (Código 12.14), que a menudo se omite. Esta cláusula incorpora aquellas instrucciones que es preciso ejecutar independientemente de si se ha generado una excepción en las instrucciones protegidas o no. Si el control alcanza el final del bloque try, entonces se saltan los bloques catch y se ejecuta la cláusula finally. Por otro lado, si se genera una excepción desde dentro del bloque try, se ejecuta el bloque catch apropiado y después se ejecuta la cláusula finally.

Código 12.14
Una instrucción try con una cláusula finally.

```
try {
    Proteger aquí una o más instrucciones.
}
catch(Exception e) {
    Informar aquí de la excepción y recuperarse de la misma.
}
finally {
    Llevar a cabo aquí todas las acciones que haya que ejecutar tanto
    si se ha generado una excepción como si no.
}
```

A primera vista, la cláusula finally parece ser redudante. ¿Acaso el siguiente ejemplo de código no ilustra el mismo fujo de control que el Código 12.14?

```
try {
    Proteger aquí una o más instrucciones.
}
catch(Exception e) {
    Informar aquí de la excepción y recuperarse de la misma.
}
    Llevar a cabo aquí todas las acciones que haya que ejecutar tanto
    si se ha generado una excepción como si no.
```

De hecho, hay al menos dos casos en los que estos dos ejemplos tendrían efectos distintos:

- La cláusula finally se ejecuta incluso aunque se ejecute una instrucción return en los bloques try o catch.
- Si se genera una excepción dentro del bloque try y esa excepción no es capturada, la cláusula finally se sigue ejecutando de todos modos.

En este último caso, la excepción no capturada podría ser una excepción no comprobada que no requiere un bloque catch, por ejemplo. Sin embargo, también podría ser una excepción comprobada que no sea tratada mediante un bloque catch, sino que se la propague desde ese método, para ser tratada a un nivel superior dentro de la pila de llamadas. En tal caso, la cláusula finally seguiría ejecutándose.

También es posible omitir los bloques catch en una instrucción try que disponga tanto de un bloque try como de una cláusula finally, si el método está propagando todas las excepciones:

```
try {
    Proteger aquí una o más instrucciones.
}
finally {
    Llevar a cabo aquí todas las acciones que haya que ejecutar tanto
    si se ha generado una excepción como si no.

}
```

12.6 Definición de nuevas clases de excepción

Cuando las clases de excepción estándar no describan satisfactoriamente la naturaleza de una condición de error, se pueden definir nuevas clases de excepción más descriptivas, haciendo uso del mecanismo de herencia. Las nuevas clases de excepción comprobada pueden definirse como subclases de cualquier clase de excepción comprobada existente (como por ejemplo `Exception`), y las nuevas excepciones no comprobadas serán subclases de la jerarquía `RuntimeException`.

Todas las clases de excepción existentes admiten la inclusión de una cadena de diagnóstico que se pasa a un constructor. Sin embargo, una de las principales razones para definir nuevas clases de excepción es incluir información adicional dentro del objeto excepción para dar soporte a tareas de diagnóstico y recuperación de errores. Por ejemplo, algunos métodos de la aplicación *address-book*, como `changeDetails`, admiten un parámetro `key` que debe corresponderse con una entrada existente. Si no es posible encontrar ninguna entrada que se corresponda, entonces esto representa un error de programación, ya que los métodos no podrán completar su tarea. A la hora de informar de la excepción, es útil incluir detalles relativos a la clave que ha provocado el error. El Código 12.15 muestra una nueva clase de excepción comprobada, definida en el proyecto *address-book-v3t*. Recibe la clave en su constructor y luego hace que esa clave esté disponible tanto a través de la cadena de diagnóstico como a través de un método selector dedicado. Si esta excepción fuera a ser capturada por una rutina de tratamiento de excepciones contenida en el llamante, la clave estaría disponible para las instrucciones que intentaran recuperarse del error.

Código 12.15
Una clase de excepción con información de diagnóstico adicional.

```
/**
 * Capturar una clave para la que no se ha encontrado
 * correspondencia con ninguna de las entradas de la
 * libreta de direcciones.
 *
 * @author David J. Barnes y Michael Kölling.
 * @version 2011.07.31
 */
public class NoMatchingDetailsException extends Exception
{
    // La clave para la que no hay correspondencia.
    private String key;

    /**
     * Almacenar los detalles erróneos.
     * @param key La clave sin correspondencia.
     */
    public NoMatchingDetailsException(String key)
    {
        this.key = key;
    }
```

Código 12.15
(*continuación*)
Una clase de excepción con información de diagnóstico adicional.

```
    /**
     * @return La clave que ha provocado el error.
     */
    public String getKey()
    {
        return key;
    }

    /**
     * @return Una cadena de diagnóstico que contiene la
     * clave que ha provocado el error.
     */
    public String toString()
    {
        return "No details matching '" + key + "' were found.";
    }
}
```

El principio de incluir información que pudiera servir de ayuda para la recuperación de errores debe tenerse presente particularmente a la hora de definir nuevas clases de excepciones comprobadas. La definición de parámetros formales en el constructor de una excepción ayudará a garantizar que haya información de diagnóstico disponible. Además, cuando la recuperación no sea posible o cuando no se intente, garantizar que se sustituya el método `toString` de la excepción para incluir información apropiada será de gran ayuda a la hora de diagnosticar la razón del error.

Ejercicio 12.33 En el proyecto *address-book-v3t*, defina una nueva clase de excepción comprobada: `DuplicateKeyException`. Esta excepción debe ser generada por el método `addDetails` si cualquiera de los campos clave de su parámetro real que no estén en blanco ya está siendo actualmente utilizado. La clase de excepción debe almacenar los detalles de la clave o claves problemáticas. Haga los cambios que sean necesarios en la clase de la interfaz de usuario para capturar e informar de la excepción.

Ejercicio 12.34 ¿Cree que `DuplicateKeyException` debe ser una excepción comprobada o no comprobada? Razone su respuesta.

12.7 Utilización de aserciones

12.7.1 Comprobaciones internas de coherencia

Cuando diseñamos o implementamos una clase, a menudo tenemos una idea intuitiva de cosas que deben ser ciertas en un determinado punto de la ejecución, aunque rara vez enunciamos esas cosas de manera formal. Por ejemplo, esperaríamos que un objeto `ContactDetails` contenga siempre al menos un campo que no esté en blanco; o bien, cuando se invoca el método `removeDetails` con una clave concreta, esperaríamos que esa clave deje de estar en

uso al finalizar el método. Normalmente, estas son condiciones que nos gustaría establecer durante el desarrollo de una clase, antes de liberar el programa para su uso comercial. En cierto sentido, los tipos de pruebas que hemos visto en el Capítulo 7 constituyen un intento de establecer si hemos implementado una representación precisa de lo que una clase o método deben hacer. Una característica de ese estilo de pruebas es que las pruebas son *externas* a la clase que está siendo probada. Si se modifica una clase, entonces debemos tomarnos el tiempo necesario para ejecutar pruebas de regresión, con el fin de comprobar que esa clase sigue funcionando como debe; sin embargo, es fácil olvidarse de hacerlo. La práctica de comprobar los parámetros, que hemos introducido en este capítulo, desplaza ligeramente el énfasis desde las pruebas completamente externas a una combinación de pruebas externas e internas. No obstante, la comprobación de parámetros está dirigida a proteger el objeto servidor frente a un uso incorrecto por parte de un cliente. Eso sigue dejándonos la cuestión de si podemos incluir algunas pruebas internas para garantizar que el objeto servidor se comporte como debe.

Una forma en que podemos implementar las pruebas internas durante el desarrollo sería a través del mecanismo normal de generación de excepciones. En la práctica, tendríamos que emplear excepciones no comprobadas, porque no podemos esperar que las clases cliente normales incluyan rutinas de tratamiento de excepciones para lo que son, esencialmente, errores internos del servidor. Entonces nos veríamos enfrentados a la cuestión de si eliminar esas pruebas internas después de haber completado el proceso de desarrollo, con el fin de evitar el coste potencialmente alto de una serie de pruebas en tiempo de ejecución que están, casi seguro, destinadas a ser pasadas con éxito.

12.7.2 La instrucción assert

Para tratar con la necesidad de realizar pruebas de coherencia internas eficientes, que puedan ser activadas en el código durante el desarrollo, pero desactivadas en el código que se libere comercialmente, hay disponible en Java una *funcionalidad de aserción*. La idea es similar a lo que vimos en el Capítulo 7 con las pruebas de JUnit, en las que enunciábamos aserciones relativas a los resultados esperados de las llamadas a los métodos, encargándose el sistema JUnit de comprobar si esas aserciones se comprobaban o no.

El proyecto *address-book-assert* es una versión de desarrollo de los proyectos *address-book* que ilustra cómo se utilizan las aserciones. El Código 12.16 muestra el método `removeDetails`, el cual ahora contiene dos formas de la *instrucción assert*.

Código 12.16
Utilización de aserciones para pruebas internas de coherencia.

```
/**
 * Eliminar de la libreta de direcciones la entrada con la clave
 * indicada. La clave debe ser una que esté actualmente en uso.
 * @param key Una de las claves de la entrada que hay que eliminar.
 * @throws IllegalArgumentException Si la clave es null.
 */
public void removeDetails(String key)
{
    if(key == null) {
        throw new IllegalArgumentException(
                "Null key passed to removeDetails.");
    }
```

Código 12.16 (*continuación*) Utilización de aserciones para pruebas internas de coherencia.

```
    if(keyInUse(key)) {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }
    assert !keyInUse(key);
    assert consistentSize() :
            "Inconsistent book size in removeDetails";
}
```

Concepto:

Una **aserción** es un enunciado de un hecho que debe ser cierto durante la ejecución del programa. Podemos utilizar aserciones para anunciar explícitamente nuestras suposiciones y para detectar más fácilmente los errores de programación.

La palabra clave `assert` va seguida por una expresión booleana. El propósito de esta instrucción es enunciar algo que debe ser cierto en ese punto del método. Por ejemplo, la primera instrucción de aserción en el Código 12.16 enuncia que `keyInUse` debe devolver `false` en ese punto, bien porque la clave no estaba siendo utilizada desde el principio o porque ya no está siendo utilizada, debido a que los detalles asociados han sido ahora eliminados de la libreta de direcciones. Esta aserción, aparentemente obvia, es más importante de lo que podría parecer a primera vista; observe que el proceso de eliminación no implica en la práctica utilizar la clave con la libreta de direcciones.

Por tanto, una instrucción de aserción tiene dos objetivos: expresar de manera explícita lo que asumimos que es cierto en un punto determinado de la ejecución, incrementando así la legibilidad tanto para el desarrollador actual como para los futuros programadores de mantenimiento; y realizar la comprobación en tiempo de ejecución, de modo que seremos notificados si nuestra suposición resulta ser incorrecta. Esto puede ser de gran ayuda para localizar los errores en una fase temprana y con mayor facilidad.

Si la expresión booleana de una instrucción de aserción se evalúa como `true`, entonces la instrucción de aserción no tiene ningún efecto ulterior. Si la instrucción se evalúa como `false`, entonces se generará un error `AssertionError`. Se trata de una subclase de `Error` (véase la Figura 12.1) y forma parte de la jerarquía que representa los errores irrecuperables –por tanto, los clientes no deben proporcionar ninguna rutina de tratamiento.

La segunda instrucción de aserción del Código 12.16 ilustra la forma alternativa de aserción. La cadena situada después de los dos puntos se pasará al constructor de `AssertionError` para proporcionar una cadena de diagnóstico. La segunda expresión no tiene por qué ser una cadena de caracteres explícita; puede emplearse cualquier expresión que proporcione un valor, y ese valor será transformado en otro de tipo `String` antes de pasárselo al constructor.

La primera instrucción de aserción ilustra que las aserciones hacen uso a menudo de uno de los métodos existentes dentro de la clase (por ejemplo, `keyInUse`). El segundo ejemplo ilustra que puede ser útil proporcionar un método con el objetivo específico de realizar una prueba de aserción (`consistentSize` en este ejemplo). Puede usarse esta técnica si la comprobación implica unos cálculos significativos. El Código 12.17 muestra el método `consistentSize`, cuyo propósito es garantizar que el campo `numberOfEntries` represente de forma precisa el número de detalles distintos contenidos en la libreta de direcciones.

Código 12.17
Comprobación de la coherencia interna de la libreta de direcciones.

```
/**
 * Comprobar que el campo numberOfEntries es coherente con el
 * número de entradas realmente almacenado en la libreta de
 * direcciones.
 * @return true si el campo es coherente y false en caso contrario.
 */
private boolean consistentSize()
{
    Collection<ContactDetails> allEntries = book.values();
    // Eliminar duplicados, ya que estamos usando múltiples claves.
    Set<ContactDetails> uniqueEntries =
            new HashSet<ContactDetails>(allEntries);
    int actualCount = uniqueEntries.size();
    return numberOfEntries == actualCount;
}
```

12.7.3 Directrices para el uso de aserciones

Las aserciones están pensadas principalmente para proporcionar un modo de realizar pruebas de coherencia durante las fases de desarrollo y pruebas de un proyecto. No están pensadas para utilizarlas en el código final para uso comercial. Es por esta razón que los compiladores Java solo incluirán las instrucciones de aserción en el código compilado si se les pide que lo hagan. De aquí se deduce que las instrucciones de aserción nunca deben utilizarse para implementar la funcionalidad normal. Por ejemplo, sería erróneo combinar las aserciones con la eliminación de detalles de la libreta de direcciones como en el ejemplo siguiente:

```
// Error: ¡no utilice assert con el procesamiento normal!
assert book.remove(details.getName()) != null;
assert book.remove(details.getPhone()) != null;
```

Ejercicio 12.35 Abra el proyecto *address-book-assert*. Examine la clase `AddressBook` e identifique todas las instrucciones de aserción, para asegurarse de que comprende lo que se está comprobando y por qué.

Ejercicio 12.36 La clase `AddressBookDemo` contiene varios métodos de prueba que invocan métodos de `AddressBook` que contienen instrucciones de aserción. Examine el código fuente de `AddressBookDemo` para cerciorarse de que comprende las pruebas, y luego experimente con cada uno de los métodos de prueba. ¿Se genera algún error de aserción? En caso afirmativo, ¿entiende por qué?

Ejercicio 12.37 El método `changeDetails` de `AddressBook` no contiene actualmente ninguna instrucción de aserción. Una aserción que podríamos enunciar acerca del mismo es que la libreta de direcciones debería contener el mismo número de entradas al final del método que al principio. Añada una instrucción de aserción (y cualquier otra instrucción que necesite) para comprobar esto. Ejecute el método `testChange` de `AddressBookDemo` después de hacerlo. ¿Cree que este método debería incluir también la comprobación de que el tamaño es coherente?

Ejercicio 12.38 Suponga que decidimos permitir que se indexe la libreta de direcciones según la dirección, además de por nombre y número de teléfono. Si nos limitamos a añadir la siguiente instrucción al método `addDetails`

```
book.put(details.getAddress(), details);
```

¿cree que fallará ahora alguna de las aserciones? Pruebe esta solución. Realice todos los cambios necesarios en `AddressBook` para garantizar que todas las aserciones se cumplan.

Ejercicio 12.39 `ContactDetails` son objetos inmutables; es decir, no tienen ningún método mutador. ¿Qué importancia tiene este hecho de cara a la coherencia interna de una libreta de direcciones `AddressBook`? Suponga que la clase `ContactDetails` tuviera, por ejemplo, un método `setPhone` para modificar el teléfono? ¿Puede desarrollar algunas pruebas que ilustren los problemas que esto podría provocar?

12.7.4 Aserciones y el entorno de prueba de unidades de BlueJ

En el Capítulo 7, hemos presentado el soporte que BlueJ proporciona para el entorno de prueba de unidades JUnit. Ese soporte se basa en la funcionalidad de definición de aserciones que hemos estado presentando en esa sección. Los métodos de ese entorno de trabajo, como por ejemplo `assertEquals`, se construyen alrededor de una instrucción de aserción que contiene una expresión booleana compuesta a partir de sus parámetros. Si se utilizan las clases de prueba JUnit para probar clases que contengan sus propias instrucciones de aserción, entonces los errores de aserción provocados por estas instrucciones serán también incluidos en la información que se presenta en la ventana de resultados de prueba, junto con los fallos de aserción que se produzcan en las clases de prueba. El proyecto *address-book-junit* contiene una clase de prueba para ilustrar esta combinación. El método `testAddDetailsError` de `AddressBookTest` provocará un error de aserción, porque `addDetails` no debería utilizarse para modificar los detalles existentes (véase el Ejercicio 12.33).

12.8 Recuperación y prevención de errores

Hasta ahora, este capítulo se ha centrado principalmente en el problema de identificar errores en un objeto servidor y garantizar que se informe de los problemas al cliente, si resulta apropiado. Hay dos cuestiones complementarias relacionadas con los informes de errores: recuperación de errores y prevención de errores.

12.8.1 Recuperación de errores

El primer requisito para una adecuada recuperación de los errores es que los clientes tomen nota de cualquier notificación de error que reciban. Esto puede parecer obvio, pero es bastante común que los programadores asuman que las llamadas a los métodos no van a fallar, y que por tanto no se preocupen de comprobar los valores de retorno. Aunque es más difícil ignorar los errores cuando se utilizan excepciones, tampoco es infrecuente encontrarse con el siguiente enfoque a la hora de tratar las excepciones:

```
AddressDetails details = null;
try {
    details = contacts.getDetails(...);
}
catch(Exception e) {
    System.out.println("Error: " + e);
}
String phone = details.getPhone();
```

La excepción ha sido capturada y se ha informado de ella, pero no se ha tenido en cuenta el hecho de que probablemente sea incorrecto continuar con el procesamiento a pesar de haberse producido la excepción.

La instrucción try de Java es la clave para suministrar un mecanismo de recuperación de errores cuando se genera una excepción. La recuperación de un error implicará usualmente llevar a cabo algún tipo de acción correctora dentro del bloque catch y luego volver a intentar la operación. Pueden realizarse intentos repetidos incluyendo la instrucción try dentro un bucle. El Código 12.18 muestra un ejemplo de esta técnica; se trata de una versión ampliada del Código 12.9. Los esfuerzos para componer un nombre de archivo alternativo podrían implicar tratar con una lista de posibles carpetas, por ejemplo, o pedir interactivamente al usuario que introduzca nombres distintos.

Código 12.18
Un intento de recuperación de errores.

```
// Tratar de guardar la libreta de direcciones.
boolean successful = false;
int attempts = 0;
do {
    try {
        addressbook.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = un nombre de archivo alternativo;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {

    Informar del problema y darse por vencido;
}
```

Aunque este ejemplo ilustra la recuperación en un situación específica, los principios involucrados tienen un carácter más general:

- Anticiparse a un error y recuperarse de él requerirá un flujo de control más complejo que si ese error no se puede producir.

- Las instrucciones del bloque catch son fundamentales para el intento de recuperación.
- La recuperación implicará a menudo tener que intentar la operación de nuevo.
- No puede garantizarse una adecuada recuperación en caso de error.
- Debe haber alguna ruta de escape que impida reintentar eternamente una recuperación imposible.

No siempre tendremos a nuestra disposición a un usuario humano para pedirle que introduzca una entrada alternativa. Puede ser responsabilidad del objeto cliente registrar el error para que este pueda ser investigado posteriormente.

12.8.2 Prevención de errores

Debería quedar claro, a estas alturas, que llegar a una situación en la que se genere una excepción será fatal, en el caso peor, para la ejecución de un programa y resultará muy lioso, en el caso mejor, recuperarse del error en el cliente. Puede ser, por tanto, mucho más simple tratar de evitar el error, aunque suele requerir una colaboración entre el servidor y el cliente.

Muchos de los casos en los que un objeto `AddressBook` se ve obligado a generar una excepción implican que se han pasado valores de parámetro `null` a sus métodos. Estos valores representan errores lógicos de programación en el cliente, que podrían obviamente evitarse aplicando previamente en el cliente una serie de pruebas simples. Los valores de parámetro nulos son normalmente el resultado de que se están haciendo suposiciones incorrectas dentro del cliente. Por ejemplo, considere el siguiente fragmento de código:

```
String key = database.search(zipCódigo);
ContactDetails university = contacts.getDetails(key);
```

Si falla la búsqueda en la base de datos, la clave que esa búsqueda devolverá podría ser un valor en blanco o `null`. Pasar ese resultado directamente al método `getDetails` hará que se genere una excepción en tiempo de ejecución. Sin embargo, utilizando una prueba simple del resultado de la búsqueda, podemos evitar la excepción y solucionar, en su lugar, el problema real subyacente, que es que la búsqueda de un cierto código postal ha fallado:

```
String key = database.search(zipCódigo);
if(key != null && key.length() > 0) {
    ContactDetails university = contacts.getDetails(key);
    . . .
}
else {
    Tratar el error relativo al código postal...
}
```

En este caso, el cliente podría determinar por sí mismo que es apropiado invocar el método del servidor. Esto no será siempre posible y en ocasiones el cliente deberá requerir la ayuda del servidor.

En el Ejercicio 12.33 hemos establecido el principio de que el método `addDetails` no debería aceptar un nuevo conjunto de detalles si uno de los valores clave ya está en uso para otro

conjunto, para evitar una llamada inapropiada, el cliente podría emplear el método `keyInUse` de la libreta de direcciones de la forma siguiente:

```
// Añadir a la libreta de direcciones lo que debería ser un
// nuevo conjunto de detalles.
if(contacts.keyInUse(details.getName()) {
    contacts.changeDetails(details.getName(), details);
}
else if(contacts.keyInUse(details.getPhone()) {
    contacts.changeDetails(details.getPhone(), details);
}
else {
    Añadir los detalles...
}
```

Utilizando este enfoque, resulta posible, obviamente, evitar por completo que `addDetails` genere una excepción `DuplicateKeyException`, lo que sugiere que esa excepción podría ser degradada de nivel, pasando de excepción comprobada a excepción no comprobada.

Este ejemplo concreto ilustra algunos principios generales importantes:

- Si los métodos de comprobación de validez y de comprobación del estado de un servidor son visibles para un cliente, este será capaz a menudo de evitar llevar a cabo acciones que hagan que el servidor genere una excepción.
- Si se puede evitar una excepción de esta forma, entonces la excepción que se está generando representa en realidad un error lógico de programación en el cliente. Esto sugiere el uso de una generación no comprobada para ese tipo de situaciones.
- La utilización de excepciones no comprobadas implica que el cliente no tiene que emplear una instrucción try cuando ya ha establecido que la excepción no va a ser generada. Esto representa una ventaja significativa, porque tener que escribir instrucciones try para situaciones que "no pueden ocurrir" es molesto para un programador, lo que a su vez hace que disminuya la probabilidad de que el programador se tome en serio la tarea de proporcionar mecanismos apropiados de recuperación para las verdaderas situaciones de error.

Sin embargo, no todos los efectos son positivos. He aquí algunas de las razones por las que este enfoque no siempre resulta práctico.

- El hacer que los métodos de comprobación de validez y de comprobación del estado de una clase servidora sean visibles públicamente para sus clientes podría representar una significativa pérdida de encapsulación y podría dar como resultado un gran acoplamiento entre el servidor y el cliente, mayor de lo deseable.
- Probablemente no sea seguro que una clase servidora presuponga que sus clientes *harán obligatoriamente* las comprobaciones necesarias que permitan evitar una excepción. Como resultado, dichas comprobaciones estarán a menudo duplicadas tanto en el cliente como en el servidor. Si esas comprobaciones son "caras" de realizar, desde el punto de vista de la capacidad de proceso, entonces dicha duplicación puede ser indeseable o prohibitiva. Sin embargo, en nuestra opinión, es mejor sacrificar la eficiencia en aras de una programación más segura, si es que podemos elegir.

12.9 Entrada/salida basada en archivo

Una importante área de la programación en la que la recuperación de errores no puede ignorarse es la de la entrada/salida. La razón es que los programadores tienen muy poco control directo sobre el entorno externo en el que se ejecuta su código. Por ejemplo, un archivo de datos necesario puede haber sido borrado accidentalmente o haberse corrompido de alguna forma antes de ejecutar la aplicación; o un intento de almacenar resultados en el sistema de archivos puede verse comprometido por la falta de los permisos adecuados o por haber excedido la cuota asignada por el sistema de archivos. Hay muchas formas en las que puede fallar en cualquier etapa una operación de entrada o de salida. Además, las técnicas modernas de entrada/salida han ido mucho más allá de los ejemplos en los que un programa simplemente se limitaba a acceder a su almacén local de archivos; ahora, nos movemos en un entorno de red en el que la conectividad con los recursos a los que se accede puede ser frágil o intermitente, por ejemplo, cuando nos encontramos en un entorno móvil.

La API Java ha experimentado una serie de evoluciones a lo largo de los años, para reflejar la creciente diversidad de entornos en los que los programas Java están siendo utilizados. El paquete fundamental relacionado con las clases de entrada/salida siempre ha sido `java.io`. Este paquete contienen numerosas clases para soportar operaciones de entrada/salida de manera independiente de la plataforma. En concreto, define la clase de excepción comprobada `IOException` como un indicador general de que algo ha ido mal en una operación de entrada/salida, y prácticamente todas las operaciones de entrada/salida deben prever que puede generarse una de estas excepciones. En ocasiones, pueden generarse instancias de subclases comprobadas de `IOException` con el fin de proporcionar información de diagnóstico más detallada, como por ejemplo `FileNotFoundException` y `EOFException`. A partir de Java 7, se está produciendo una migración desde una serie de clases del paquete `java.io` a las de la jerarquía `java.nio`, aunque sin que haya quedado completamente obsoleto todo lo que contiene `java.io`. Vamos a presentar algunas de las nuevas clases junto con las antiguas.

Una descripción completa de la multitud de distintas clases existentes en los paquetes `java.io` y `java.nio` queda fuera del alcance de este libro, pero proporcionaremos algunos ejemplos fundamentales, dentro del contexto de varios de los proyectos que ya hemos visto. Esto debería proporcionar al lector la suficiente información básica como para poder experimentar con la entrada/salida en sus propios proyectos. En particular, vamos a ilustrar las siguientes tareas comunes:

- Obtener del sistema de archivos información acerca de un archivo.
- Escribir salida textual en un archivo con la clase `FileWriter`.
- Leer entrada textual de un archivo con las clases `FileReader` y `BufferedReader`.
- Prever las excepciones `IOException`.
- Analizar sintácticamente la entrada con la clase `Scanner`.

Además, veremos cómo leer y escribir versiones binarias de los objetos, como breve introducción a la funcionalidad de *serialización* de Java.

Para obtener más información sobre la entrada/salida en Java, le recomendamos el tutorial de Oracle, al que se puede acceder en línea en la dirección:

```
http://download.oracle.com/javase/tutorial/essential/io/index.html
```

12.9.1 Lectores, escritores y flujos de datos

Varias de las clases del paquete `java.io` caen dentro de dos categorías principales: las que tratan con archivos de texto y las que manejan archivos binarios. Podemos pensar en los archivos de texto como si contuvieran datos en un formato similar al tipo `char` de Java –típicamente, se tratara de información alfanumérica simple con estructura de líneas y que además es legible para los seres humanos. Las páginas web, escritas en HTML, son un ejemplo concreto de este tipo de archivos. Los archivos binarios tienen una mayor variedad: los archivos de imagen son un ejemplo común, como también lo son los programas ejecutables como por ejemplo los procesadores de texto y los reproductores multimedia. Las clases de Java dedicadas al procesamiento de archivos de texto se conocen con el nombre de *lectores* y *escritores*, mientras que las que manejan archivos binarios se conocen con el nombre de clases para tratamiento de *flujos*. Aquí nos vamos a centrar fundamentalmente en los lectores y escritores.

12.9.2 La clase `File` y la interfaz `Path`

Un archivo es mucho más que simplemente un nombre y un cierto contenido. Por ejemplo, un archivo puede estar ubicado en una *carpeta* o en un *directorio* concretos dentro de una unidad de disco determinada, y los diferentes sistemas operativos utilizan distintos convenios relativos a los caracteres que se pueden utilizar en los nombres de ruta de los archivos. La clase `File` permite a un programa consultar los detalles relativos a un archivo externo, de una forma independiente del sistema de archivos concreto sobre el que se esté ejecutando el programa. El nombre del archivo se pasa al constructor de `File`. Crear un objeto `File` dentro de un programa no crea un archivo dentro del sistema de archivos; más bien, lo que hace es que se almacenen en el objeto `File` los detalles acerca de un archivo, si es que ese archivo externo existe.

En Java 7, la interfaz `Path` de `java.nio.file` cumple un papel similar. Puesto que `Path` es una interfaz y no una clase, las instancias concretas de las clases que la implementan se crean a través de un método `get` estático de la clase `Paths` (observe el nombre en plural), que se encuentra también en el paquete `java.nio.file`. Cuando se trabaja con código antiguo, puede crearse un objeto `Path` equivalente a partir de un objeto `File` mediante el método `toPath` de `File`.

En ocasiones, podemos evitar meternos en situaciones que requieran tratar una excepción, utilizando un objeto `File` o un objeto `Path` para comprobar si un archivo existe o no. Un objeto `File` dispone de sendos métodos `exists` y `canRead`, que hacen que el intentar abrir un archivo tenga menos probabilidades de fallar si primero utilizamos esos métodos. Sin embargo, como abrir un archivo que sabemos que existe *sigue requiriendo* que se incluya el mecanismo para tratar una potencial excepción, la mayoría de los programadores no se molestan en hacer esas comprobaciones previas.

`Path` no dispone por sí misma de métodos equivalentes. En lugar de ello, la clase `Files` (de nuevo en plural) proporciona un gran número de métodos estáticos para consultar los atributos de un objeto `Path`; por ejemplo, `exists`, `isReadable`, `isDirectory`, etc.

Ejercicio 12.40 Lea la documentación de la API para la clase `File` del paquete `java.io`. ¿Qué tipo de información hay disponible acerca de los archivos?

Ejercicio 12.41 Utilizando un objeto `File`, ¿cómo podemos decir si un nombre de archivo representa un archivo normal o un directorio (carpeta)?

Ejercicio 12.42 ¿Es posible determinar alguna información acerca del contenido de un archivo concreto a partir de la información almacenada en un objeto `File`?

Ejercicio 12.43 Si está utilizando Java 7, repita los tres ejercicios anteriores empleando las clases `Path` y `Files`.

12.9.3 Salida a través de archivo

El proceso de almacenamiento de datos en un archivo consta de tres pasos:

1. Se abre el archivo.
2. Se escriben los datos.
3. Se cierra el archivo.

La naturaleza de la salida a través de archivo implica que cualquiera de estos pasos podría fallar por diversas razones, muchas de las cuales caen completamente fuera del control del programador de la aplicación. Como consecuencia, será necesario prever que se puedan generar excepciones en cualquiera de las etapas.

Para escribir un archivo de texto es habitual crear un objeto `FileWriter`, cuyo constructor acepta el nombre del archivo que se va a escribir. El nombre del archivo puede estar en la forma de un objeto `String` o de un objeto `File`. Crear un `FileWriter` tiene el efecto de abrir el archivo externo y prepararlo para recibir una cierta información de salida. Si el intento de abrir el archivo fallara por cualquier razón, entonces el constructor generaría una `IOException`. Las razones de un posible fallo pueden ser que los permisos del sistema de archivos impidan a un cierto usuario escribir en determinados archivos, o que el nombre de archivo especificado no se corresponda con una ubicación válida dentro del sistemas de archivos.

Cuando se ha abierto un archivo adecuadamente, entonces pueden utilizarse los métodos `write` del escritor para almacenar caracteres en el archivo –a menudo en forma de cadenas. Cualquier intento de escribir podría fallar, incluso si el archivo se ha abierto correctamente. Dichos fallos son raros, pero continúan siendo posibles.

Una vez escrita toda la información de salida, es importante cerrar formalmente el archivo. Esto garantizará que todos los datos se escriban realmente en el sistema de archivos externo y a menudo también tiene el efecto de liberar algunos recursos internos o externos. De nuevo, en algunas raras ocasiones ese intento de cerrar el archivo podría fallar.

El patrón básico al que nos llevan las explicaciones anteriores tiene el siguiente aspecto:

```
try {
    FileWriter writer = new FileWriter("... nombre de archivo ...");
    while(haya más texto que escribir) {
        ...
         writer.write(siguiente fragmento de texto);
        ...
    }
```

```
        writer.close();
    }
    catch(IOException e) {
        algo ha ido mal al tratar con el archivo
    }
```

El problema principal que surge es cómo tratar las excepciones que se generen durante las tres etapas. Las únicas excepciones con las que probablemente pueda hacerse algo son las generadas al intentar abrir un archivo, y solamente podrá hacerse algo si existe alguna forma de generar un nombre alternativo con el que reintentar la operación. Como esto requerirá, normalmente, la intervención de un usuario humano de la aplicación, las posibilidades de tratar adecuadamente con esa excepción son, obviamente, específicas de la aplicación y específicas del contexto. Si falla un intento de escribir en el archivo, entonces es muy poco probable que repetir ese intento permita completar la operación con éxito. De forma similar, no suele merecer la pena efectuar un reintento cuando se produce un fallo a la hora de cerrar un archivo. La consecuencia más probable es que tengamos un archivo incompleto.

En el Código 12.19 podemos ver un ejemplo de este patrón de comportamiento. La clase `LogfileCreator` del proyecto *weblog-analyzer* incluye un método para escribir una serie de entradas de registro aleatorias en un archivo, cuyo nombre se pasa como parámetro al método `createFile`. Este utiliza dos métodos `write` distintos: uno para escribir una cadena y otro para escribir un carácter. Después de escribir el texto de la entrada en forma de cadena, escribimos un carácter de avance de línea para que cada entrada aparezca en una línea diferente.

Código 12.19
Escritura en un archivo de texto.

```
/**
 * Crear un archivo con entradas de registro aleatorias.
 * @param filename El archivo en el que hay que escribir.
 * @param numEntries Cuántas entradas.
 * @return true si tiene éxito, false en caso contrario.
 */
public boolean createFile(String filename, int numEntries)
{
    boolean success = false;
    try {
        FileWriter writer = new FileWriter(filename);
        LogEntry[] entries = new LogEntry[numEntries];
        for(int i = 0; i < numEntries; i++) {
            entries[i] = createEntry();
        }
        Arrays.sort(entries);
        for(int i = 0; i < numEntries; i++) {
            writer.write(entries[i].toString());
            writer.write('\n');
        }

        writer.close();
        success = true;
    }
```

Código 12.19 (*continuación*) Escritura en un archivo de texto.

```
    catch(IOException e) {
        System.err.println("There was a problem writing to " +
                           filename);
    }
    return success;
}
```

Un problema con el patrón mostrado en el Código 12.19 es que si se produce un fallo después de haber abierto el archivo, ese archivo se quedará abierto. Esto hace que se desperdicie una pequeña cantidad de los recursos del programa, pero lo más importante es que confunde al sistema de archivos, porque este ve obligado a asumir que el programa sigue utilizando el archivo. Lea la siguiente sección para ver una forma de tratar con las situaciones de error como consecuencia de los cambios introducidos en Java 7.

12.9.4 La instrucción try con recursos

Java 7 introdujo en la instrucción try una característica para tratar con aquellas situaciones en las que un determinado recurso, como por ejemplo un archivo abierto, debe obligatoriamente cerrarse una vez que se ha terminado de utilizar –independientemente de si el uso de ese recurso se completó con éxito o no. Esta nueva funcionalidad se denomina *try con recursos* o *gestión automática de recursos* (ARM). El Código 12.20 muestra la manera de emplear esta funcionalidad utilizando una versión alternativa del Código 12.19; puede encontrar esta versión en el proyecto *weblog-analyzer-v7*.

Código 12.20 Escritura en un archivo de texto utilizando una instrucción try con recurso.

```
/**
 * Crear un archivo con entradas de registro aleatorias..
 * @param filename El archivo en el que hay que escribir.
 * @param numEntries Cuántas entradas.
 * @return true si tiene éxito, false en caso contrario.
 */
public boolean createFile(String filename, int numEntries)
{
    boolean success = false;
    try(FileWriter writer = new FileWriter(filename)) {
        Se omite la creación de la entrada

        for(int i = 0; i < numEntries; i++) {
            writer.write(entries[i].toString());
            writer.write('\n');
        }
        success = true;
    }
    catch(IOException e) {
        System.err.println("There was a problem writing to " +
                           filename);
    }
    return success;
}
```

El recurso se crea dentro de nueva sección entre paréntesis, inmediatamente después de la palabra `try`. Una vez completada la instrucción try, se invocará automáticamente el método `close` para ese recurso. Esta versión de la instrucción try solo es apropiada para los objetos de las clases que implementan la interfaz `AutoCloseable`, definida en el paquete `java.lang`. Normalmente, se tratará de clases asociadas con la entrada/salida.

Ejercicio 12.44 Modifique el proyecto *world-of-zuul* para que escriba un registro de las entradas del usuario en un archivo de texto, con el fin de ir guardando un registro del juego. Hay varias formas diferentes en las que se nos podría ocurrir llevar a cabo esta tarea:

- Podríamos modificar la clase `Parser` para almacenar cada línea de entrada en una lista y luego escribir la lista al final del juego. Esto nos dará una solución que será la más parecida al patrón básico que hemos esbozado en las explicaciones anteriores.
- Podríamos incluir todo lo relativo al tratamiento de archivos en la clase `Parser`, de modo que, a medida que va siendo leída cada línea, esta se escriba inmediatamente, exactamente de la misma forma en que fue leída. La apertura, la escritura y el cierre del archivo estarán separados en el tiempo unos de otros (¿o acaso tendría sentido abrir el archivo, escribir y luego cerrar el archivo para cada línea que se escriba?).
- Podríamos incluir todo lo relativo al tratamiento de archivos en la clase `Game` e implementar un método `toString` en la clase `Command`, que devuelva el objeto `String` que haya que escribir para cada comando.

Considere cada una de estas posibles soluciones en términos de las directrices del diseño dirigido por responsabilidad y del tratamiento de excepciones, junto con las posibles implicaciones a la hora de jugar con el programa, si resulta imposible escribir el registro de acciones del usuario. ¿Cómo podemos garantizar que el archivo de registro se cierre siempre una vez alcanzado el final del juego? Esto es importante para garantizar que toda la información de salida se escriba realmente en el sistema de archivos externo.

12.9.5 Entrada de texto

El complemento de la salida de texto con un `FileWriter` es la entrada mediante un `FileReader`. Como cabría esperar, hace falta un conjunto complementario de los tres pasos de entrada: abrir el archivo, leerlo y cerrarlo. Al igual que las unidades naturales para escribir texto son los caracteres y las cadenas, las unidades más obvias para la lectura de texto son los caracteres y las líneas. Sin embargo, aunque la clase `FileReader` contiene un método para leer un único carácter,[6] no contiene un método para leer una línea. El problema con la lectura de líneas de un archivo es que no existe ningún límite predefinido en lo que respecta a la longitud de la línea. Esto significa que cualquier método que devuelva la siguiente línea completa de un archivo debe ser capaz de leer un número arbitrario de caracteres. Por esta razón, normalmente es conveniente trabajar con la clase `BufferedReader`, que define un método `readLine`. Sin embargo, ¡`BufferedReader` no puede abrir archivos! Por tanto, debemos crear primero un `FileReader` e inmediatamente envolverlo en un objeto `BufferedReader`. Después, podemos ignorar el objeto `FileReader` y trabajar solamente con el `BufferedReader`.

[6] De hecho, su método `read` devuelve cada carácter como un valor `int` en lugar de como `char`, porque utiliza un valor fuera de límites, -1, para indicar el final del archivo. Este es exactamente el tipo de técnica que describimos anteriormente en la Sección 12.3.2.

El carácter de fin de línea siempre se elimina del objeto `String` devuelto por `readLine`, utilizándose un valor `null` para indicar el final del archivo.

Esto sugiere el siguiente esquema básico para leer el contenido de un archivo de texto:

```
try {
    BufferedReader reader = new BufferedReader(
                            new FileReader("... nombre de archivo..."));
    String line = reader.readLine();
    while(line != null) {
        hacer algo con la línea
        line = reader.readLine();
    }
    reader.close();
}
catch(FileNotFoundException e) {
    No se ha podido encontrar el archivo especificado
}
catch(IOException e) {
    Algo fue mal con la lectura o el cierre del archivo
}
```

Observe que ni siquiera nos molestamos en almacenar el objeto `FileReader` en una variable, sino que se lo pasamos directamente al constructor de `BufferedReader`. Esto evita la confusión de tener dos variables `Reader` haciendo referencia al mismo tipo de entrada, porque debemos acceder al archivo únicamente a través del objeto `BufferedReader`.

Con Java 7, como antes, utilizaríamos aquí una instrucción try con recursos, pero vea también a continuación una forma alternativa de crear el `BufferedReader` a través de la clase `Files`.

El Código 12.21 ilustra la utilización práctica de un `BufferedReader` en el proyecto *tech-support* del Capítulo 5. Esto se ha hecho para poder leer las respuestas predeterminadas del sistema desde un archivo, en lugar de codificarlas directamente en el código de la clase `Responder` (proyecto: *tech-support-io*).

Código 12.21
Lectura de un archivo de texto.

```
import java.io.*;
import java.util.*;

public class Responder
{
    // Respuestas predeterminadas que hay que utilizar si no
    // reconocemos una palabra.
    private List<String> defaultResponses;
    // El nombre del archivo que contiene las respuestas
    // predeterminadas.
    private static final String FILE_OF_DEFAULT_RESPONSES =
                                "default.txt";
```

Código 12.21
(*continuación*)
Lectura de un archivo de texto.

```
    ... Se omiten otros campos y métodos...

    /**
     * Construir una lista de respuestas predeterminadas entre las
     * que podamos elegir si no sabemos qué otra cosa decir.
     */
    private void fillDefaultResponses()
    {
        try {
            BufferedReader reader = new BufferedReader(
                        new FileReader(FILE_OF_DEFAULT_RESPONSES));
            String response = reader.readLine();
            while(response != null) {
                defaultResponses.add(response);
                response = reader.readLine();
            }
            reader.close();
        }
        catch(FileNotFoundException e) {
            System.err.println("Unable to open " +
                               FILE_OF_DEFAULT_RESPONSES);
        }
        catch(IOException e) {
            System.err.println("A problem was encountered reading "
                               + FILE_OF_DEFAULT_RESPONSES);
        }
        // Asegurarse de tener al menos una respuesta.
        if(defaultResponses.size() == 0) {
            defaultResponses.add("Could you elaborate on that?");
        }
    }
}
```

Como con la salida, surge la cuestión de qué hacer con las excepciones que se generen durante todo el proceso. En este ejemplo, hemos impreso un mensaje de error y luego hemos proporcionado al menos una respuesta en caso de que falle completamente el intento de leer algo.

Con Java 7, la forma normal de crear un `BufferedReader` es a través del método estático `newBufferedReader` de la clase `Files`. Además de un parámetro `Path` correspondiente al archivo que hay que abrir, una complicación es que también hace falta un parámetro `Charset`. Este se utiliza para describir el conjunto de caracteres al que pertenecen los caracteres del archivo. `Charset` puede encontrarse en el paquete `java.nio.charset`. Existen una serie de conjuntos de caracteres estándar, como `US-ASCII` e `ISO-8859-1`, y puede encontrar más información en la documentación de la API para `Charset`. El Código 12.22 muestra el uso de estas características para leer un archivo de texto.

Código 12.22
Lectura de un archivo de texto con Java 7.

```
import java.io.*;
import java.nio.charset.Charset;
import java.nio.file.*;
import java.util.*;

public class Responder
{
    ... campos y métodos omitidos...

    /**
     * Construir una lista de respuestas predeterminadas entre las
     * que podamos elegir si no sabemos qué otra cosa decir.
     */
    private void fillDefaultResponses()
    {
        Charset charset = Charset.forName("US-ASCII");
        Path path = Paths.get(FILE_OF_DEFAULT_RESPONSES);
        try(BufferedReader reader =
                Files.newBufferedReader(path, charset)) {
            String response = reader.readLine();
            while(response != null) {
                defaultResponses.add(response);
                response = reader.readLine();
            }
        }
        catch(FileNotFoundException e) {
            System.err.println("Unable to open " +
                                   FILE_OF_DEFAULT_RESPONSES);
        }
        catch(IOException e) {
            System.err.println("A problem was encountered reading "
+
                                   FILE_OF_DEFAULT_RESPONSES);
        }
        // Asegurarse de tener al menos una respuesta.
        if(defaultResponses.size() == 0) {
            defaultResponses.add("Could you elaborate on that?");
        }
    }
}
```

Ejercicio 12.45 El archivo `default.txt` de la carpeta de proyectos contiene las respuestas predeterminadas leídas por el método `fillDefaultResponses`. Utilizando cualquier editor de texto, modifique el contenido de este archivo para que haya una línea vacía entre cada dos respuestas. Después, cambie su código para que vuelva a funcionar correctamente al leer las respuestas de este archivo.

Ejercicio 12.46 Modifique su código para que los bloques de líneas de código que haya en el archivo y que no estén separados por una línea vacía se lean como una única respuesta. Cambie el archivo `default.txt` para que contenga varias respuestas que abarquen múltiples líneas. Prueba la solución desarrollada.

Ejercicio 12.47 Modifique la clase `Responder` del proyecto *tech-support-io* de modo que lea las asociaciones entre palabras clave y respuestas desde un archivo de texto, en lugar de inicializar `responseMap` con cadenas escritas en el código fuente del método `fillResponseMap`. Puede utilizar el método `fillDefaultResponses` como patrón, pero tendrá que hacer algunos cambios en su lógica interna, porque ahora hay que leer dos cadenas de caracteres para cada entrada (la palabra clave y la respuesta) en lugar de solo una. Trate de almacenar las palabras clave y las respuestas en líneas alternativas; mantenga el texto de cada respuesta en una única línea. Puede suponer que siempre habrá un número par de líneas (es decir, que no falta ninguna respuesta).

12.9.6 Scanner: análisis sintáctico de la entrada

Hasta ahora, hemos tratado la entrada fundamentalmente como una serie de líneas de texto no estructuradas, para las que la clase `BufferedReader` es la ideal. Sin embargo, lo que muchas aplicaciones harán a continuación es descomponer las líneas individuales en sus partes componentes, que representarán valores de datos de múltiples tipos. Por ejemplo, el formato de valores separados por comas (CSV, *Comma-separated values*) se utiliza comúnmente para almacenar archivos de texto cuyas líneas están compuestas por múltiples valores, estando cada uno de ellos separado de los valores adyacentes mediante un carácter de coma.[7] En la práctica, tales líneas de texto tienen una estructura implícita. La identificación de la estructura subyacente se conoce con el nombre de *análisis sintáctico,* mientras que descomponer los caracteres individuales en valores de datos separados se conoce como *escaneo*.

La clase `Scanner` del paquete `java.util` está específicamente diseñada para analizar sintácticamente el texto y convertir secuencias compuestas de caracteres en valores con un determinado tipo, como enteros (`nextInt`) y números en coma flotante (`nextDouble`). Aunque un `Scanner` puede utilizarse para descomponer objetos `String`, también se emplea a menudo para leer y convertir directamente el contenido de archivos en lugar de usar `BufferedReader`. Dispone de constructores que pueden admitir argumentos de tipo `String`, `File`, o (en Java 7) `Path`. El Código 12.23 ilustra cómo se puede leer de esta manera un archivo de texto que haya que interpretar como compuesto por datos enteros.

Observe que este método no garantiza que se lea el archivo completo. El método `hasNextInt` de `Scanner` que controla el bucle devolverá `false` si encuentra texto en el archivo que no parezca ser parte de un número entero. En ese punto, se terminará el proceso de recopilación de datos. De hecho, es perfectamente posible mezclar llamadas a los diferentes métodos `next` a la hora de analizar sintácticamente un archivo completo, cuyos datos deban ser interpretados como compuestos por una mezcla de tipos.

---

[7] En la práctica, puede utilizarse cualquier carácter en lugar de una coma; por ejemplo, también se emplea frecuentemente un carácter de tabulación.

Código 12.23
Lectura de datos enteros con `Scanner`.

```
/**
 * Leer enteros de un archivo y devolverlos
 * en forma de matriz.
 * @param filename El archivo que hay que leer.
 * @return Los enteros leídos.
 */
public int[] readInts(String filename)
{
    int[] data;
    try {
        List<Integer> values = new ArrayList<Integer>();
        Scanner scanner = new Scanner(new File(filename));
        while(scanner.hasNextInt()) {
            values.add(scanner.nextInt());
        }
        // Copiarlos a una matriz del tamaño exacto.
        data = new int[values.size()];
        Iterator<Integer> it = values.iterator();
        int i = 0;
        while(it.hasNext()) {
            data[i] = it.next();
            i++;
        }
    }
    catch(FileNotFoundException e) {
        System.out.println("Cannot find file: " + filename);
        data = new int[0];
    }
    return data;
}
```

Otro uso común de `Scanner` es el de leer la entrada desde el "terminal" conectado a un programa. Hemos utilizado regularmente llamadas a los métodos `print` y `println` de `System.out` para escribir texto en la ventana de terminal de BlueJ. `System.out` es de tipo `java.io.PrintStream` y se corresponde con lo que a menudo se denomina destino de *salida estándar*. Correspondientemente, hay un origen de *entrada estándar* disponible como `System.in`, que es de tipo `java.io.InputStream`. Cuando hace falta leer entrada del usuario desde el terminal, normalmente no se usa directamente un `InputStream`, porque suministra la entrada de carácter en carácter. En lugar de ello, lo que se suele hacer es pasar `System.in` al constructor de un `Scanner`. La clase `InputReader` del proyecto *tech-support-complete* del Capítulo 5 utiliza esta técnica para leer las preguntas del usuario:

```
Scanner reader = new Scanner(System.in);
... código intermedio omitido...
String inputLine = reader.nextLine();
```

El método `nextLine` de `Scanner` devuelve la siguiente línea completa de entrada desde la entrada estándar (sin incluir el carácter final de avance de línea).

Ejercicio 12.48 Repase la clase `InputReader` de *tech-support-complete* para comprobar que entiende cómo se emplea allí la clase `Scanner`.

Ejercicio 12.49 Lea la documentación de la API para la clase `Scanner` en el paquete `java.util`. ¿Qué otros métodos `next` tiene además de los que hemos explicado en esta sección?

Ejercicio 12.50 Repase la clase `Parser` de *zuul-better* para ver también cómo se usa allí la clase `Scanner`. Se utiliza de dos formas ligeramente distintas.

Ejercicio 12.51 Repase la clase `LoglineTokenizer` de *weblog-analyzer* para ver cómo utiliza un `Scanner` para extraer los valores enteros de las líneas de registro.

12.9.7 Serialización de objetos

Concepto:

La **serialización** permite leer y escribir en una única operación objetos completos, así como jerarquías de objetos. Todos los objetos implicados deben pertenecer a alguna clase que implemente la interfaz `Serializable`.

En términos sencillos, la serialización permite escribir un objeto completo en un archivo externo en una única operación de escritura y leerlo posteriormente mediante una única operación de lectura.[8] Esto funciona tanto con objetos simples como con objetos multicomponente, como por ejemplo las colecciones. Se trata de una funcionalidad muy importante, que evita tener que leer y escribir objetos campo a campo. Es particularmente útil en aplicaciones que dispongan de datos persistentes, como por ejemplo libretas de direcciones o bases de datos de información multimedia, porque permite que todas las entradas creadas en una sesión se guarden y se lean más tarde en una sesión posterior. Por supuesto, tenemos la opción de escribir el contenido de los objetos como cadenas de texto, pero el proceso de volver a leer el texto, convertir los datos a los tipos correctos y restaurar *el estado exacto* de un conjunto complejo de objetos es a menudo difícil, sino imposible. La serialización de objetos es un proceso mucho más fiable.

Para que una clase sea eligible de cara a participar en una serialización deberá implementar la interfaz `Serializable` que está definida en el paquete `java.io`. Sin embargo, merece la pena resaltar que esta interfaz no define ningún método. Esto quiere decir que el proceso de serialización es gestionado automáticamente por el sistema de tiempo de ejecución y requiere escribir muy poco código definido por el usuario. En el proyecto *address-book-io*, tanto `AddressBook` como `ContactDetails` implementan esta interfaz, para poder guardar esos objetos en un archivo. La clase `AddressBookFileHandler` define los métodos `saveToFile` y `readFromFile` para ilustrar el proceso de serialización. El Código 12.24 contiene el código fuente de `saveToFile` para ilustrar que en realidad hace falta muy poco código para guardar la libreta de direcciones completa, en una única instrucción de escritura. Observe también que, debido a que estamos escribiendo objetos en formato binario, hemos empleado un objeto `Stream` en lugar de un `Writer`. La clase `AddressBookFileHandler` también incluye ejemplos adicionales de las técnicas básicas de lectura y escritura utilizadas con los archivos de texto. Examine, por ejemplo, sus métodos `saveSearchResults` y `showSearchResults`.

---

[8] Esto es una simplificación, porque los objetos pueden escribirse y leerse también a través de una red, por ejemplo, y no solo dentro de un sistema de archivos.

Código 12.24
Serialización de un objeto `AddressBook` completo con todos los detalles de `ContactDetails`.

```
public class AddressBookFileHandler
{
    ... campos y métodos omitidos...
    /**
     * Guardar una versión binaria de la libreta de direcciones
     * en el archivo indicado. Si el nombre de archivo no es una
     * ruta absoluta, entonces se supone que es relativa a la
     * carpeta de proyecto actual.
     * @param destinationFile El archivo en el que hay que
     *                        guardar los detalles.
     * @throws IOException Si el proceso de guardado falla
     *                     por cualquier razón.
     */
    public void saveToFile(String destinationFile) throws
IOException
    {
        File destination = makeAbsoluteFilename(destinationFile);
        ObjectOutputStream os = new ObjectOutputStream(
                                   new FileOutputStream
                                       (destination));
        os.writeObject(book);
        os.close();
    }
}
```

Ejercicio 12.52 Modifique el proyecto *network* del Capítulo 9 para que los datos puedan almacenarse en un archivo. Utilice para ello el mecanismo de serialización de objetos. ¿Qué clases tendrá que declarar que sean serializables?

Ejercicio 12.53 ¿Qué sucede si modifica la definición de una clase añadiendo, por ejemplo, un campo adicional y luego trata de leer objetos serializados creados a partir de la versión previa de esa clase?

12.10 Resumen

Cuando dos objetos interactúan, siempre existe la posibilidad de que algo pueda ir mal, por diversas razones. Por ejemplo:

- El programador de un cliente podría haber entendido mal el estado o las capacidades de un objeto servidor concreto.
- Un objeto servidor puede ser incapaz de satisfacer una solicitud de un cliente, debido a un conjunto concreto de circunstancias externas.
- Un cliente podría haber sido programado incorrectamente, haciendo que pase valores de parámetro inapropiados a un método servidor.

Si algo llega a ir mal, es probable que el programa termine de forma prematura (es decir, que se produzca un fallo catastrófico) o que produzca efectos incorrectos e indeseados. Son muchas las cosas que podemos hacer para tratar de evitar muchos de estos problemas, utilizando el mecanismo de generación de excepciones. Este mecanismo proporciona una forma claramente definida para que un objeto informe a un cliente de que algo ha salido mal. Las excepciones impiden que el cliente se limite a ignorar el problema y animan a los programadores a tratar de encontrar formas alternativas de actuación como solución a los problemas, si es que algo sale mal.

A la hora de desarrollar una clase pueden utilizarse instrucciones de aserción para disponer de comprobaciones internas de coherencia. Esas instrucciones de aserción normalmente se omiten en el código de producción.

La entrada/salida es una área en la que hay bastante probabilidad de que se generen excepciones. Esto se debe principalmente a que un programador tiene muy poco control (si es que tiene alguno) sobre los entornos en los que se ejecutan sus programas, pero también refleja la complejidad y la diversidad de esos entornos de ejecución de programas.

La API de Java soporta la entrada/salida de datos tanto de texto como binarios, a través de lectores, escritores y flujos de datos. Java 7 ha introducido una serie de cambios en la E/S, con la incorporación de los paquetes `java.nio`.

Ejercicio 12.54 Añada al proyecto *address-book* la capacidad de almacenar varias direcciones de correo electrónico en un objeto `ContactDetails`. Todas esas direcciones de correo electrónico deberán ser claves válidas. Utilice aserciones y pruebas JUnit en todas las etapas de este proceso para adquirir la máxima confianza en la versión final.

Términos introducidos en el capítulo

excepción, excepción no comprobada, excepción comprobada, rutina de tratamiento de excepciones, aserción, serialización

Resumen de conceptos

- **excepción** Una excepción es un objeto que representa los detalles de un fallo del programa. La excepción se genera para indicar que se ha producido un fallo.
- **excepción no comprobada** Una excepción no comprobada es un tipo de excepción cuyo uso no requiere ninguna comprobación por parte del compilador.
- **excepción comprobada** Una excepción comprobada es un tipo de excepción cuyo uso requiere comprobaciones adicionales por parte del compilador. En particular, las excepciones comprobadas en Java requieren que se usen cláusulas throws e instrucciones try.
- **rutina de tratamiento de excepciones** El código de programa que protege las instrucciones en las que podría generarse una excepción se denomina rutina de tratamiento de excepción. Proporciona código para informar de la excepción y/o recuperarse de la misma, en caso de que se genere una.

- **aserción** Una aserción es un enunciado de un hecho que debe ser cierto durante la ejecución del programa. Podemos utilizar aserciones para anunciar explícitamente nuestras suposiciones y para detectar más fácilmente los errores de programación.
- **serialización** La serialización permite leer y escribir en una única operación objetos completos, así como jerarquías de objetos. Todos los objetos implicados deben pertenecer a alguna clase que implemente la interfaz `Serializable`.

CAPÍTULO

13 Diseño de aplicaciones

Principales conceptos explicados en el capítulo:

- descubrimiento de clases
- tarjetas CRC
- diseño de interfaces
- patrones

Estructuras Java explicadas en este capítulo:

(En este capítulo no se presenta ninguna nueva estructura Java.)

En los capítulos anteriores de este libro, hemos descrito cómo escribir buenas clases. Hemos hablado de cómo diseñarlas, de cómo hacerlas mantenibles y robustas, y de cómo conseguir que interactúen. Todo esto es importante, pero hemos omitido un aspecto crucial de la tarea: identificar las clases que tenemos que desarrollar.

En todos los ejemplos anteriores, hemos asumido que conocíamos más o menos qué clases podríamos emplear para resolver nuestros problemas. Pero en un proyecto real de software, decidir qué clases utilizar para implementar una solución a un problema puede ser una de las tareas más difíciles. En este capítulo, vamos a analizar este aspecto del proceso de desarrollo.

Estos pasos iniciales de desarrollo de un sistema software se suelen denominar, generalmente, *análisis y diseño.* Analizamos el problema y luego diseñamos una solución. El primer paso del diseño estará, por tanto, a un nivel más alto del diseño de clases del que hemos hablado en el Capítulo 6. Tenemos que pensar en qué clases tenemos que crear para solucionar nuestro problema y en cómo exactamente deben interactuar. Una vez que hayamos resuelto esta cuestión, podremos continuar con el diseño de las clases individuales y empezar a pensar en su implementación.

13.1 Análisis y diseño

El análisis y diseño de sistemas software es un área muy amplia y compleja. Hablar de ella en detalle cae fuera del alcance de este libro. A lo largo de los años, se han descrito en la literatura científica muchas metodologías distintas y una gran cantidad de ellas se emplean en la práctica para esta tarea. En este capítulo únicamente pretendemos hacer una introducción a los tipos de problemas que podemos encontrarnos durante el proceso.

Utilizaremos un método relativamente sencillo para abordar estas tareas, que sirve perfectamente para problemas relativamente pequeños. Para descubrir las clases iniciales, utilizaremos el *método de los verbos/nombres*. Después utilizaremos *tarjetas CRC* para llevar a cabo el diseño inicial de la aplicación.

13.1.1 El método de los verbos/nombres

Concepto:

verbos/nombres Las clases de un sistema se corresponden aproximadamente con los nombres existentes en la descripción del sistema. Los métodos se corresponden con los verbos.

Este método trata de identificar clases y objetos, así como las asociaciones e interacciones entre ellos. Los nombres en un lenguaje humano describen "cosas", como por ejemplo gente, edificios, etc. Los verbos describen "acciones", como por ejemplo escribir o comer. A partir de estos conceptos del lenguaje natural, podemos ver que en la descripción de un problema de programación, los nombres a menudo se corresponderán con clases y objetos, mientras que los verbos se corresponderán con las cosas que esos objetos pueden hacer, es decir, los métodos. No necesitamos una descripción muy farragosa para poder ilustrar esta técnica. La descripción solo necesita, normalmente, tener unos cuantos párrafos de longitud.

El ejemplo que utilizaremos para explicar este proceso es el diseño de un sistema de reserva de entradas de cine.

13.1.2 El ejemplo de la reserva de entradas de cine

Esta vez no vamos a comenzar ampliando un proyecto existente. Ahora vamos a suponer que nos encontramos en una situación en la que nos han encargado crear una nueva aplicación partiendo de cero. La tarea consiste en crear un sistema que una empresa propietaria de salas de cine puede emplear para gestionar las reservas de entradas para las películas. Hay mucha gente que llama de antemano para reservar sus entradas. Por tanto, la aplicación deberá ser capaz de localizar asientos vacíos en un determinado pase de película y reservarlos para el cliente.

Asumiremos que hemos celebrado diversas reuniones con la empresa propietaria de las salas de cine, durante las que la empresa nos ha descrito la funcionalidad que espera obtener del sistema. (Comprender cuál es la funcionalidad esperada, describirla y llegar a un acuerdo sobre la misma con el cliente es un problema que tiene bastante complejidad en sí mismo. Sin embargo, esto cae fuera del alcance de este libro y puede estudiarse en otros cursos y otros libros de texto.)

He aquí la descripción que hemos escrito para nuestro sistema de reserva de entradas:

> *El sistema de reserva de entradas de cine debe almacenar reservas de asientos para varias salas. Cada sala tiene una serie de asientos dispuestos en filas. Los clientes pueden reservar asientos y se les proporciona un número de fila y un número de asiento. Pueden solicitar que se reserven varios asientos adyacentes.*
>
> *Cada reserva es para un pase concreto de película (es decir, para la proyección de una cierta película en un cierto momento). Los pases tienen una fecha y una hora asignadas y están programados para una determinada sala, en la que se proyecta la película. El sistema almacena el número de teléfono del cliente.*

Dada una descripción razonablemente clara como esta, podemos realizar un primer intento de descubrir las clases y métodos identificando los nombres y los verbos en el texto.

13.1.3 Descubrimiento de las clases

El primer paso a la hora de identificar las clases consiste en repasar la descripción y marcar todos los nombres y verbos contenidos en el texto. Al hacer esto con la descripción de nuestro problema encontramos los siguientes nombres y verbos. (Los nombres se muestran en el orden en que aparecen en el texto; los verbos aparecen asociados con los nombres a los que hacen referencia.

Nombres	Verbos
sistema de reserva de entradas	*almacena* (reservas de asiento) *almacena* (número de teléfono)
reserva de asiento	
sala	*tiene* (asientos)
asiento	
fila	
cliente	*reserva* (asientos) *recibe* (número de fila, número de asiento) *solicita* (reserva de asiento)
número de fila	
número de asiento	
pase	*está programado* (en una sala)
película	
fecha	
hora	
número de teléfono	

Los nombres que hemos identificado aquí nos dan una primera aproximación a las clases de nuestro sistema. Como primera solución podemos utilizar una clase para cada nombre. Este no es un método exacto; posteriormente podemos encontrarnos con que necesitamos algunas clases adicionales o con que algunos de los nombres no son necesarios. Sin embargo, esto lo comprobaremos un poco más adelante. Es importante no excluir ningún nombre desde el principio; no disponemos todavía de la suficiente información como para tomar una decisión razonable.

Casi siempre cuando hacemos este ejercicio con nuestros estudiantes, algunos de ellos inmediatamente prescinden de algunos nombres. Por ejemplo, muchos estudiantes prescinden del nombre *fila* contenido en la descripción anterior. Cuando se les pregunta por qué lo hacen, suelen contestar: "Bueno, eso es simplemente un número, así que me limitaré a utilizar un entero, eso no necesita que se defina una clase". Sin embargo, es muy importante no hacer este tipo de cosas en esta etapa. Realmente no tenemos la suficiente información en este punto como para decidir si la *fila* debe ser un `int` o una clase. Solo mucho más adelante podremos tomar ese tipo de decisiones. Por ahora, lo que hacemos es recorrer el párrafo mecánicamente extrayendo *todos* los nombres, sin hacer ningún juicio todavía sobre cuáles son los "adecuados" y cuáles no.

Observe que todos los nombres se han escrito en singular. Es bastante habitual que los nombres de las clases se expresen en singular en lugar de en plural. Por ejemplo, siempre decidiremos

definir una clase como `Cinema` en lugar de como `Cinemas`. La razón es que la multiplicidad se consigue al crear múltiples instancias de una clase.

Ejercicio 13.1 Repase los proyectos de los capítulos anteriores de este libro. ¿Hay algún caso de algún nombre de clase que sea plural? En caso afirmativo, ¿están justificadas esas situaciones por alguna razón concreta?

13.1.4 Utilización de tarjetas CRC

El siguiente paso en nuestro proceso de diseño consiste en determinar las interacciones entre las clases. Para hacer esto, utilizaremos un método denominado *tarjetas CRC.*[1]

CRC significa *Class/Responsibilities/Collaborators* (Clase/Responsabilidades/Colaboradores). La idea es tomar una serie de tarjetas y utilizar cada una de ellas para cada clase. Es importante llevar a cabo esta actividad utilizando tarjetas físicas reales, no simplemente una computadora o una única hoja de papel. Cada tarjeta se divide en tres áreas: una en la parte superior izquierda, en la que se escribe el nombre de la clase, otra área debajo de esta para anotar las responsabilidades de la clase y un área a la derecha para escribir los colaboradores de esta clase (clases que serán utilizadas por esta). La Figure 13.1 ilustra la disposición de una tarjeta CRC.

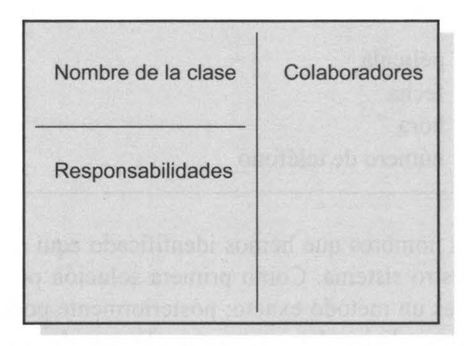


Figura 13.1
Una tarjeta CRC.

Ejercicio 13.2 Prepare unas tarjetas CRC para las clases del sistema de reserva de entradas de cine. En esta etapa, lo único que tiene que completar son los nombres de las clases.

13.1.5 Escenarios

Ahora tenemos una primera aproximación a las clases que hacen falta en nuestro sistema, así como una representación física de las mismas en tarjetas CRC. Para determinar las

[1] Las tarjetas CRC fueron descritas por primera vez en un artículo de Kent Beck y Ward Cunningham, titulado *A Laboratory For Teaching Object-Oriented Thinking*. Merece la pena leer dicho artículo como información suplementaria a este capítulo. Puede encontrarlo en línea en la dirección `http://c2.com/doc/oopsla89/paper.html` o buscando su título en la Web.

Concepto:

Pueden utilizarse **escenarios** (también conocidos como "casos de uso") para comprender las interacciones dentro de un sistema.

interacciones necesarias entre las clases de nuestro sistema, lo que hacemos es jugar con una serie de *escenarios*. Un escenario es un ejemplo de una actividad que el sistema tiene que llevar a cabo o tiene que soportar. Los escenarios también se denominan *casos de uso*, pero nosotros no vamos a utilizar dicho término aquí, porque se suele emplear para designar una manera más formal de describir los escenarios.

La mejor manera de jugar con los escenarios es en grupo. A cada miembro del grupo se le asigna una clase (o un pequeño número de clases) y dicha persona desempeña su papel diciendo en voz alta lo que la clase está haciendo actualmente. Mientras se juega con el escenario, la persona anota en la tarjeta CRC todo lo que se va averiguando acerca de la actuación de esa clase: cuáles deberían ser sus responsabilidades y con qué otras clases colabora.

Vamos a comenzar con un escenario sencillo de ejemplo:

> Un cliente llama al cine y dice que desea hacer una reserva de dos entradas para esta noche, para ver la película clásica *The Shawshank Redemption*. El empleado del cine comienza a utilizar el sistema de reservas para localizar y reservar un asiento.

Puesto que el usuario humano interactúa con el sistema de reservas (representado por la clase `CinemaBookingSystem`), es aquí donde comienza el escenario. Por ejemplo, lo que podría suceder a continuación es lo siguiente:

- El usuario (el empleado del cine) quiere encontrar todos los pases de *The Shawshank Redemption* para esta noche. Por tanto, podemos anotar como responsabilidad en la tarjeta CRC de `CinemaBookingSystem` lo siguiente: *puede localizar los pases por título y día.* También podemos anotar la clase `Show` , que representa los pases, como colaborador.
- Tenemos que preguntarnos a nosotros mismos: ¿cómo localiza el sistema los pases? ¿A quién le pregunta? Una solución podría ser que la clase `CinemaBookingSystem` almacene una colección de pases. Esto nos daría una clase adicional: la colección. (Esto puede implementarse posteriormente utilizando un `ArrayList`, un `LinkedList`, un `HashSet` o algún otro tipo de colección. Podemos tomar esta decisión más tarde; por ahora nos limitaremos a anotar esto como una `Collection`.) Este es un ejemplo de cómo se pueden introducir clases adicionales durante la experimentación con los escenarios. Puede suceder en determinados momentos que tengamos que añadir clases por razones de implementación que inicialmente habíamos pasado por alto. Añadiremos a las responsabilidades de la tarjeta de `CinemaBookingSystem`: *almacena colecciones de pases*. Y añadiremos `Collection` a los colaboradores.

Ejercicio 13.3 Haga una tarjeta CRC para la nueva clase `Collection` identificada y añádala a su sistema.

- Suponemos que se localizan tres pases: uno a las 5:30 p.m., otro a las 9:00 p.m. y otro a las 11:30 p.m. El empleado informa al cliente de las horas de los pases y el cliente selecciona el de las 9:00 p.m. Por tanto, el empleado quiere ahora comprobar los detalles de dicho pase (si están agotadas las entradas, en qué sala tiene lugar el pase, etc.). Por tanto, en nuestro sistema, la clase `CinemaBookingSystem` deberá ser capaz de extraer y mostrar los detalles del pase. Tratemos de representar esto. La persona que hace el papel del sistema de reservas debe pedir a la persona que hace el papel del pase que le proporcione los detalles requeridos. Entonces, anotaremos en la tarjeta correspondiente a `CinemaBookingSystem`: *extrae y muestra los detalles del pase*, mientras que en la tarjeta correspondiente a `Show` escribiríamos: *proporciona los detalles acerca de la sala y el número de asientos libres.*

- Suponga que hay muchos asientos libres. El cliente selecciona los asientos 13 y 14 de la fila 12. El empleado hace esa reserva. Anotamos en la tarjeta de `CinemaBookingSystem`: *acepta reservas de asientos por parte del usuario.*

- Ahora tenemos que reproducir cómo funciona exactamente el sistema de reserva de asientos. Cada reserva de asiento está claramente asociada a un pase concreto, por lo que `CinemaBookingSystem` probablemente deberá informar al pase acerca de la reserva y delegar la tarea real de hacer la reserva al objeto `Show`. Podemos anotar en la clase `Show`: *puede reservar asientos*. (Puede que haya observado que la diferenciación entre objetos y clases está un poco difuminada a la hora de jugar con los escenarios CRC. De hecho, la persona que representa una clase representa también a sus instancias. Esto es intencionado y no suele plantear ningún problema.)

- Ahora es el turno de la clase `Show`. Ha recibido una solicitud para reservar un asiento. ¿Qué es exactamente lo que tendrá que hacer? Para poder almacenar reservas de asientos, tiene que tener una representación de los asientos existentes en la sala. Por tanto, asumiremos que cada pase dispone de un enlace a un objeto `Theater`, que representará a la sala. (Anote esto en la tarjeta: *almacena salas*. La sala será también un colaborador.) La sala debería probablemente saber también acerca del número de asientos que tiene y su disposición. (También podemos anotar mentalmente, o en una hoja de papel separada, que cada pase debe tener su propia instancia del objeto `Theater`, porque puede haber varios pases programados para un mismo objeto `Theater` y reservar un asiento en uno de ellos no debe hacer que se reserve el mismo asiento para otros pases. Esto es algo de lo que tendremos que encargarnos cuando creemos objetos `Show`. Tenemos que tener esto en mente para que luego, cuando juguemos con el escenario de *Planificación de un nuevo pase*, nos acordemos de crear una instancia de sala para cada pase.) La forma en que un pase maneja la reserva de un asiento consistirá, probablemente, en pasar esa solicitud de reserva a la sala.

- Ahora la sala ha aceptado una solicitud para hacer una reserva. (Anote esto en la tarjeta: *acepta solicitudes de reserva*.) ¿Cómo gestiona esto? La sala podría tener una colección de asientos. O bien podría tener una colección de filas (cada una de las cuales sería un objeto distinto), mientras que las filas a su vez contienen asientos. ¿Cuál de estas alternativas es mejor? Tratando de anticiparnos a otros posibles escenarios, podríamos decidirnos por la solución consistente en almacenar filas. Por ejemplo, si un cliente solicita cuatro asientos juntos en la misma fila, puede ser más fácil localizar cuatro asientos consecutivos si tenemos todos los asientos organizados por filas. Anotaremos en la tarjeta de `Theater`: *almacena filas*. `Row`, que es la clase que representará las filas, es ahora un colaborador.

- En la clase `Row` anotamos: *almacena una colección de asientos*. Y luego anotamos un nuevo colaborador: `Seat`, que representará los asientos.

- Volviendo a la clase `Theater`, todavía no hemos resuelto cómo debe reaccionar exactamente a la solicitud de reserva de un asiento. Vamos a suponer que hace dos cosas: localizar la fila solicitada y luego realizar una solicitud de reserva al objeto `Row`, utilizando el número de asiento especificado.

- A continuación, anotamos en la tarjeta `Row`: *acepta solicitudes de reserva de asientos*. Deberá encontrar el objeto `Seat` correcto (podemos anotar esto como responsabilidad: *puede localizar asientos por su número*) y puede llevar a cabo la reserva de dicho asiento. Para ello, lo que haría es decir al objeto `Seat` que ahora está reservado.

- Ahora podemos añadir a la tarjeta de `Seat`: *acepta reservas*. El propio asiento puede recordar si ha sido reservado. Anotamos en la tarjeta de `Seat`: *almacena el estado de reserva (libre/reservado)*.

Ejercicio 13.4 Juegue con este escenario utilizando sus propias tarjetas (si es posible, con un grupo de personas). Añada cualquier otra información que crea que hayamos olvidado en esta descripción.

¿Debería el asiento almacenar información acerca de quién lo ha reservado? Podría almacenar el nombre o el número de teléfono del cliente. ¿O quizá deberíamos crear, en cuanto alguien haga una reserva, un objeto `Customer` que represente al objeto y almacenar el objeto `Customer` con el asiento, después de haberlo reservado? Son cuestiones interesantes y trataremos de determinar cuál es la mejor solución jugando con otros escenarios adicionales.

Esto era simplemente un primer escenario sencillo. Tenemos que experimentar con muchos otros escenarios para intentar comprender cómo debería funcionar el sistema.

El reproducir escenarios funciona mejor cuando hay un grupo de personas sentado alrededor de una mesa y mueven las tarjetas de un sitio a otro. Las tarjetas que cooperen estrechamente pueden situarse juntas, para captar mejor el grado de acoplamiento existente en el sistema.

Otros escenarios con los que jugar a continuación serían, por ejemplo:

- Un cliente solicita cinco asientos consecutivos. Determine exactamente cómo se pueden localizar cinco asientos juntos.
- Un cliente llama y dice que ha olvidado los números de asiento que le dieron cuando ayer hizo su reserva. ¿Podría por favor consultar de nuevo cuáles son los números de asiento?
- Un cliente llama para cancelar una reserva. Puede facilitar su nombre y el pase, pero se ha olvidado de los números de asiento.
- Llama un cliente que ya tiene una reserva. Quiere saber si puede reservar otro asiento que esté junto a los que ya ha reservado.
- Se cancela un pase. La empresa propietaria del cine quiere llamar a todos los clientes que han reservado un asiento para ese pase.

Estos escenarios deberían proporcionarle una buena comprensión de la parte del sistema relativa a la búsqueda y reserva de asientos. A continuación, necesitamos otro grupo de escenarios: aquellos que se ocupan de la configuración de las salas y de la programación de los pases. He aquí algunos posibles escenarios:

- Hay que preparar el sistema para incorporar un nuevo complejo de salas. El complejo tiene dos salas de tamaños diferentes. La sala A tiene 26 filas con 18 asientos cada una. La sala B tiene 32 filas. En esta sala, las primeras seis filas tienen 20 asientos, las diez filas siguientes tienen 22 asientos y las filas restantes tienen 26 asientos.
- Queremos programar una nueva película para su proyección. Se proyectará durante las siguientes dos semanas, tres veces al día (a las 4:40 p.m., las 6:30 p.m. y las 8:30 p.m.). Hay que añadir los pases al sistema. Todos los pases tendrán lugar en la sala A.

Ejercicio 13.5 Experimente con estos escenarios. Anote en una hoja de papel separada todas las preguntas que haya dejado sin responder. Haga un registro de todos los escenarios con los que haya jugado.

Ejercicio 13.6 ¿Qué otros escenarios podría imaginar? Descríbalos y luego experimente con ellos.

Jugar con escenarios requiere algo de paciencia y de práctica. Es importante dedicar el tiempo suficiente a hacer esto. Experimentar con los escenarios que hemos mencionado aquí requerirá varias horas.

Es muy común que los programadores principiantes tomen atajos y no cuestionen ni anoten cada detalle relativo a la ejecución de un escenario. ¡Eso es muy peligroso! Pronto pasaremos a la fase de desarrollar este sistema en Java, y si dejamos detalles sin responder, es muy probable que tengamos que tomar decisiones sobre la marcha durante la implementación, decisiones que pueden luego resultar erróneas.

También es común que los principiantes se olviden de algunos escenarios. Olvidarse de pensar en una parte del sistema antes de iniciar el diseño e implementación de las clases puede dar lugar a una buena cantidad de trabajo posteriormente, cuando haya que cambiar un sistema que ya está parcialmente implementado.

Llevar a cabo correctamente esta actividad, recorriendo con cuidado todos los pasos necesarios y anotando esos pasos con el suficiente detalle, requiere algo de práctica y un montón de disciplina. Este ejercicio es bastante más complicado de lo que parece y mucho más importante de lo que se imagina.

Ejercicio 13.7 Haga un diseño de clases para una simulación de un sistema de control aeroportuario. Utilice tarjetas CRC y escenarios. He aquí una descripción del sistema:

El programa es un sistema de simulación aeroportuaria. Para nuestro nuevo aeropuerto, necesitamos determinar si podemos operar con dos pistas o necesitamos tres. El aeropuerto funciona de la forma siguiente:

Hay varias pistas. Los aviones despegan y aterrizan en las pistas. Los controladores aéreos coordinan el tráfico y dan a los aviones permiso para despegar o aterrizar. En ocasiones, los controladores dan el permiso directamente, pero en otros casos les dicen a los aviones que esperen. Los aviones deben mantener una cierta distancia entre sí. El propósito del programa es simular el funcionamiento del aeropuerto.

13.2 Diseño de clases

Ahora es el momento de dar el siguiente gran paso: pasar de las tarjetas CRC a las clases Java. Durante el ejercicio de las tarjetas CRC debería haber podido comprender adecuadamente cómo se estructura su aplicación y cómo sus clases cooperan para resolver las tareas que el programa tiene asignadas. Puede que se haya encontrado con casos en los que haya tenido que introducir clases adicionales (esto suele pasar con clases que representen estructuras internas de datos), y puede también haberse encontrado con que tiene alguna tarjeta para una clase que nunca ha sido utilizada. En este último caso, esta tarjeta puede ahora eliminarse.

Reconocer las clases para la implementación es ahora algo trivial. Las tarjetas nos muestran el conjunto completo de clases que necesitamos. Decidir acerca de la interfaz de cada clase (es decir, del conjunto de métodos públicos que una clase debe tener) es algo más difícil, pero

hemos dado un paso importante en ese sentido. Si el juego con los escenarios se ha hecho correctamente, entonces las responsabilidades anotadas para cada clase describen los métodos públicos de esa clase (y quizá algunos de los campos de instancia). Las responsabilidades de cada clase deben evaluarse de acuerdo con los principios de diseño de clases expuestos en el Capítulo 6: diseño dirigido por responsabilidad, acoplamiento y cohesión.

13.2.1 Diseño de interfaces de clases

Antes de comenzar a codificar nuestra aplicación en Java, podemos utilizar una vez más las tarjetas para dar otro paso más hacia el diseño final, traduciendo las descripciones informales en llamadas a métodos y añadiendo los parámetros necesarios.

Para conseguir esas descripciones más formales, podemos ahora volver a jugar con los escenarios, pero esta vez hablando en términos de llamadas a métodos, parámetros y valores de retorno. La lógica y la estructura de la aplicación ya no deben cambiar, pero tenemos que intentar anotar información lo más completa posible sobre las signaturas de los métodos y los campos de instancia. Haremos esto en un nuevo conjunto de tarjetas.

Ejercicio 13.8 Haga un nuevo conjunto de tarjetas CRC para las clases que haya identificado. Experimente de nuevo con los escenarios. Esta vez anote los nombres exactos de los métodos para cada método que invoque desde otra clase y especifique en detalle (con su nombre y su tipo) todos los parámetros pasados y los valores de retorno de los métodos. Las signaturas de los métodos se escriben en la tarjeta CRC en lugar de las responsabilidades. En la parte posterior de la tarjeta, anote los campos de instancia que tiene cada clase.

Una vez hecho el ejercicio anterior, es sencillo escribir la interfaz de cada clase. Podemos traducir directamente de las tarjetas a Java. Normalmente, deberíamos crear todas las clases y escribir *esqueletos de métodos* para los métodos públicos. Un esqueleto de método es un sustituto del método que tiene la signatura correcta y un cuerpo de método vacío.[2]

Muchos estudiantes encuentran que hacer esto de forma detallada es tedioso. Sin embargo, al final del proyecto, lo más probable es que llegue a apreciar lo mucho que ayudan estas actividades. Muchos equipos de desarrollo software se han dado cuenta demasiado tarde de que el tiempo ahorrado durante la etapa de diseño tuvo luego que ser invertido muchas más veces para corregir errores u omisiones que no se descubrieron lo suficientemente temprano.

Los programadores inexpertos a menudo ven la escritura de código como la "verdadera programación". Llevar a cabo las tareas de diseño inicial es visto como algo, si no superfluo, al menos molesto y esos programadores casi son incapaces de esperar a terminar con esa tarea, para poder empezar lo antes posible con el "trabajo real". Esta es una manera de pensar completamente errónea.

El diseño inicial es una de las partes más importantes del proyecto. Debemos planificar pensando en invertir al menos tanto tiempo trabajando en el diseño como en la implementación. El diseño de aplicaciones no es algo que preceda a la programación –¡*Es* (la parte más importante de) la programación!

[2] Si lo desea, puede incluir instrucciones de retorno triviales en los cuerpos de los métodos, con tipos de retorno distintos de `void`. Simplemente devuelva un valor `null` en los métodos que devuelvan objetos y un valor cero, o `false`, e n el caso de tipos primitivos.

Los errores en el propio código pueden corregirse posteriormente de manera bastante sencilla, pero los errores en el diseño global pueden, como poco, ser muy caros de corregir y, en el caso peor, pueden resultar fatales para la aplicación en su conjunto. En los casos más desafortunados, pueden ser casi imposibles de corregir (empezando todo de nuevo).

13.2.2 Diseño de interfaces de usuario

Un aspecto que hemos dejado fuera de nuestras explicaciones hasta el momento es el diseño de la interfaz de usuario.[3] En algún momento tendremos que decidir con detalle qué es lo que los usuarios verán en la pantalla y cómo van a interactuar con nuestro sistema.

En una aplicación bien diseñada, esto es bastante independiente de la lógica subyacente de la aplicación, así que puede realizarse de forma independiente del diseño de la estructura de clases para el resto del proyecto. Como hemos visto en los capítulos anteriores, BlueJ nos proporciona el modo de interactuar con nuestra aplicación antes de que esté disponible una interfaz de usuario final, por lo que podemos decidir trabajar primero en la estructura interna.

La interfaz de usuario puede ser una GUI (*graphical user interface*) con menús y botones, o puede estar basada en texto, o podemos decidir ejecutar la aplicación utilizando el mecanismo de llamada a métodos de BlueJ. Quizá el sistema vaya a ejecutarse a través de una red y la interfaz de usuario se presente en un explorador web en una máquina distinta.

Por ahora, ignoraremos el diseño de la interfaz de usuario y utilizaremos la invocación de métodos en BlueJ para trabajar con nuestro programa.

13.3 Documentación

Después de identificar las clases y sus interfaces, y antes de empezar a implementar los métodos de una clase, es necesario documentar la interfaz. Esto implica escribir un comentario para la clase y los comentarios de los métodos para cada clase del proyecto. Estos comentarios deben incluir los suficientes detalles como para poder identificar el propósito global de cada clase y método.

Junto con el análisis y el diseño, la documentación es otra de las áreas que a menudo es ignorada por los principiantes. No resulta fácil para los programadores inexpertos ver por qué la documentación es tan importante. La razón es que los programadores inexpertos suelen trabajar en proyectos que solo cuentan con un puñado de clases y que se escriben en el plazo de unas pocas semanas o meses. Cuando se trabaja con estos mini-proyectos, los programadores pueden perfectamente convivir con una documentación defectuosa.

Sin embargo, incluso los programadores experimentados se preguntan a menudo cómo es posible escribir la documentación antes de la implementación. Esto se debe a que no se dan cuenta de que una buena documentación se centra en las cuestiones de alto nivel, como por

[3] Observe cuidadosamente el doble significado que aquí tiene el término "diseño de interfaces". Hasta ahora, en el capítulo hablábamos de las interfaces de las clases individuales (un conjunto de métodos públicos); ahora, hablamos de la *interfaz de usuario* –lo que el usuario ve en la pantalla para interactuar con la aplicación. Ambas cosas son muy importantes y lamentablemente se utiliza el término *interfaz* para ambas.

ejemplo en qué es lo que hace una clase o un método, más que en cuestiones de bajo nivel como son los detalles exactos de cómo llevar a cabo su tarea. Esto suele ser sintomático de que esos programadores piensan que la implementación es más importante que el diseño.

Si un desarrollador de software quiere progresar para abordar problemas más interesantes y comienza a trabajar profesionalmente en aplicaciones del mundo real, no será inusual que tenga que trabajar con docenas de otras personas en una misma aplicación a lo largo de varios años. La solución *ad hoc* de "tener la documentación simplemente en la cabeza" ya no sirve en absoluto.

Ejercicio 13.9 Cree un proyecto BlueJ para el sistema de reserva de entradas de cine. Cree las clases necesarias. Cree esqueletos de método para todos los métodos.

Ejercicio 13.10 Documente todas las clases y métodos. Si ha estado trabajando en grupo, asigne la responsabilidad de cada clase a diferentes miembros de grupo. Utilice el formato `javadoc` para los comentarios, empleando marcadores `javadoc` apropiados para documentar los detalles.

13.4 Cooperación

Programación en pareja Los programadores suelen trabajar solos para implementar las clases. La mayoría de los programadores trabaja a su aire mientras escriben el código, y solo se involucran otras personas una vez que la implementación se ha terminado, para probar o revisar el código.

Más recientemente se ha sugerido la programación en pareja como alternativa que pretende producir un código de mejor calidad (un código con mejor estructura y menor número de errores). La programación en pareja es también uno de los elementos de una técnica conocida como *Programación Extrema*. Busque en la Web los términos *pair programming* o *extreme programming* para obtener más información.

El desarrollo de software suele hacerse en equipo. Un enfoque limpio y orientado a objetos sirve de mucha ayuda a la hora de trabajar en equipo, porque permite separar el problema en componentes débilmente acoplados (clases) que se pueden implementar de forma independiente.

Aunque la mejor forma de realizar el trabajo de diseño inicial era en grupo, ahora es el momento de dividirse. Si se ha hecho bien la definición de las interfaces de las clases y la documentación, debería ser posible implementar las clases de forma independiente. Las clases ahora se pueden asignar a programadores que pueden trabajar sobre ellas solos o en pareja.

En el resto de este capítulo, no vamos a analizar en detalle la fase de implementación del sistema de reserva de entradas de cine. Esa fase implica, en buena medida, el tipo de tareas que hemos estado haciendo a lo largo de todos los capítulos anteriores, y esperamos que llegados a este punto los lectores puedan determinar por sí mismos cómo continuar a partir de aquí.

13.5 Prototipado

En lugar de diseñar y luego construir la aplicación completa en un único paso de gigante, puede utilizarse el *prototipado* para investigar parte de un sistema.

Concepto:

El **prototipado** es la construcción de un sistema parcialmente funcional, en el que algunas funciones de la aplicación están simuladas. Sirve para tratar de comprender en una fase temprana del proceso de desarrollo cómo funcionará el sistema.

Un prototipo es una versión de la aplicación en la que se simula una parte con el fin de experimentar con otras partes. Podemos, por ejemplo, implementar un prototipo para probar una interfaz gráfica de usuario. En ese caso, puede que la lógica de la aplicación no esté apropiadamente implementada. En lugar de ello, deberíamos escribir implementaciones simples para aquellos métodos que simulen la tarea. Por ejemplo, al invocar un método para localizar un asiento libre en el sistema de reserva de entradas, el método podría siempre devolver *asiento 3, fila 15* en lugar de realizar verdaderamente la búsqueda. El prototipado nos permite desarrollar un sistema ejecutable (aunque no completamente funcional) de forma rápida, para poder investigar de manera práctica determinadas partes de la aplicación.

Los prototipos también son útiles para clases individuales, como ayuda a un proceso de desarrollo en grupo. A menudo, cuando los diferentes miembros del equipo trabajan con distintas clases, no todas las clases tardan lo mismo en ser completadas. En algunos casos, una clase que falte puede impedir que se continúe con el desarrollo y las pruebas de otras clases. En esos casos, puede ser ventajoso escribir un prototipo de esa clase. El prototipo tiene implementaciones para todos los esqueletos de método, pero en lugar de contener implementaciones completas y finales, el prototipo solo simula la funcionalidad. Debería ser posible escribir el prototipo rápidamente, con lo que el desarrollo de las clases cliente puede entonces continuar, empleando el prototipo de manera interina, hasta que la clase esté implementada.

Como se ha explicado en la Sección 13.6, una ventaja adicional del prototipado es que puede proporcionar al desarrollador información adicional acerca de cuestiones y problemas que no se hubieran tenido en cuenta en las etapas anteriores.

Ejercicio 13.11 Esboce un prototipo para el ejemplo del sistema de las salas de cine. ¿Cuáles de las clases deberían implementarse primero y cuáles deberían permanecer en estado de prototipo?

Ejercicio 13.12 Implemente su prototipo del sistema de las salas de cine.

13.6 Crecimiento del software

Existen varios modelos para la construcción de software. Uno de los más conocidos es el que a menudo se denomina *modelo en cascada* (porque la actividad va progresando de un nivel al siguiente como el agua que va cayendo por una cascada –no hay vuelta atrás).

13.6.1 Modelo en cascada

En el modelo en cascada se llevan a cabo varias fases del desarrollo software siguiendo una secuencia fija:

- Análisis del problema.
- Diseño del software.
- Implementación de los componentes software.
- Prueba de unidades.

- Pruebas de integración.
- Entrega del sistema al cliente.

Si falla cualquiera de las fases, puede que tengamos que volver a la fase anterior para corregir el fallo (por ejemplo, si las pruebas muestran algún error, volvemos atrás a la implementación), pero nunca existe un plan de volver deliberadamente a las fases anteriores.

Probablemente, este es el modelo más tradicional y conservador de desarrollo software, y ha estado siendo utilizado ampliamente durante un largo tiempo. Sin embargo, a lo largo de los años se han ido descubriendo numerosos problemas que aquejan a este modelo. Dos de los fallos principales son que presupone que los desarolladores comprenden completa y detalladamente desde el principio el alcance de la funcionalidad del sistema, y que el sistema no va a sufrir modificaciones después de haber sido entregado al cliente.

En la práctica, ambas suposiciones suelen ser falsas. Es bastante común que el diseño de la funcionalidad de un sistema no sea perfecto al principio. Esto se debe a menudo a que el cliente, que conoce el dominio del problema, no sabe mucho acerca de programación, y a que los ingenieros software que saben cómo programar, solo tienen un conocimiento limitado del dominio del problema.

13.6.2 Desarrollo iterativo

Una posibilidad de resolver los problemas del modelo en cascada consiste en utilizar el mecanismo de prototipado rápido y una serie de frecuentes interacciones con el cliente durante el proceso de desarrollo. Se construyen prototipos de los sistemas que no hacen mucho, pero que proporcionan una idea acerca del aspecto que tendrá el sistema y de lo que va a hacer, y permiten a los clientes realizar comentarios periódicos acerca del diseño y la funcionalidad. Esto lleva a un proceso de carácter más circular que el del modelo en cascada. Aquí, el desarrollo software va iterando varias veces a través del ciclo *análisis–diseño–implementación de prototipo–realimentación de cliente*.

Otra solución está representada por la noción de que un software de calidad no se diseña, sino que *crece*. La idea que subyace a este concepto consiste en diseñar inicialmente un sistema pequeño y limpio y conseguir que sea completamente funcional, para poder ser empleado por los usuarios finales. Después, se van añadiendo gradualmente funciones adicionales (el software crece) de una forma controlada, y se van alcanzando de forma repetida y frecuente una serie de estados "finales" (es decir, estados en los que el software es completamente utilizable y puede ser entregado a los clientes).

En realidad, hacer crecer el software no está, por supuesto, en contradicción con el diseño de software. Cada paso de crecimiento se diseña con sumo cuidado. Lo que el modelo de crecimiento del software no intenta hacer es diseñar todo el sistema completo desde el principio. De hecho, ¡la noción de un sistema de software completo ni siquiera existe dentro de este modelo!

El modelo en cascada tradicional tiene como objetivo el suministro de un sistema completo. El modelo de crecimiento del software asume que no existen sistemas completos que se utilicen indefinidamente en un estado inmutable. Solo hay dos cosas que pueden pasar con un sistema software: o se mejora y adapta de manera continua o desaparece.

Estos conceptos son fundamentales en lo que a este libro se refiere, porque influyen fuertemente en nuestra forma de contemplar las tareas y en las capacidades que se le exigen

a un programador o a un ingeniero software. Probablemente, se haya dado cuenta de que los autores de este libro somos fervientes partidarios del modelo de crecimiento del software más que del modelo en cascada.[4]

Como consecuencia, ciertas tareas y capacidades son mucho más importantes de lo que serían con el modelo en cascada. El mantenimiento del software, la lectura de código (en lugar de solo la escritura), el diseño con vistas a la ampliabilidad, la documentación, la codificación con vistas a la legibilidad y muchas otras cuestiones que hemos mencionado en el libro cobran importancia por el hecho de que somos conscientes de que habrá otros programadores que vengan después de nosotros que tengan que adaptar y ampliar nuestro código.

Contemplar un software como una entidad que está continuamente creciendo, cambiando y adaptándose, en lugar de como un fragmento estático de texto que se escribe y se conserva como una novela, influye decisivamente sobre nuestra visión acerca de cómo debe escribirse un buen código. Todas las técnicas de las que hemos hablado a lo largo del libro están orientadas en esta dirección.

Ejercicio 13.13 ¿De qué formas podría adaptarse o ampliarse en el futuro el sistema de reserva de entradas de cine? ¿Qué cambios son más probables que otros? Escriba a continuación una lista de los posibles futuros cambios.

Ejercicio 13.14 ¿Hay alguna otra organización que pudiera utilizar sistemas de reserva similares al que hemos estado analizando? ¿Qué diferencias significativas existen entre esos sistemas?

Ejercicio 13.15 ¿Cree que sería posible diseñar un sistema de reservas "genérico" que pudiera adaptarse o personalizarse para ser empleado en un amplio rango de organizaciones distintas, que compartan la necesidad de realizar reservas? Si fuera a crear tal sistema, ¿en qué punto del proceso de desarrollo del sistema de las salas de cine introduciría cambios? ¿O tiraría todo el diseño y comenzaría de nuevo partiendo de cero?

13.7 Uso de patrones de diseño

Concepto:

Un **patrón de diseño** es una descripción de un problema común de programación, así como una descripción de un pequeño conjunto de clases (junto con su estructura de interacción) que ayuda a resolver el problema.

En capítulos anteriores, hemos explicado detalladamente algunas técnicas para reutilizar parte de nuestro trabajo y para hacer que nuestro código sea más fácilmente compresible para otras personas. Hasta hora, buena parte de estos análisis se referían al código fuente de clases individuales.

A medida que vayamos ganando experiencia y pasemos a diseñar sistemas software de mayor envergadura, la implementación de clases individuales deja de ser el problema más difícil. La estructura del sistema global –las complejas relaciones entre las clases– se hace más difícil de diseñar y de comprender que el código de las clases individuales.

Resulta evidente que deberíamos tratar de conseguir para las estructuras de clases los mismos objetivos que perseguíamos para el código fuente. Queremos reutilizar buena parte del trabajo y queremos también permitir que otros puedan entender lo que hemos hecho.

---

[4] Un excelente libro que describe los problemas del desarrollo de software y algunos posibles enfoques de solución es *The Mythical Man-Month* de Frederick P. Brooks Jr., Addison-Wesley. Aunque la edición original tiene casi 40 años, su lectura es muy entretenida y clarificadora.

En el nivel de las estructuras de clases, ambos objetivos pueden alcanzarse utilizando *patrones de diseño*. Un patrón de diseño describe un problema común que aparece de manera habitual durante el desarrollo software y luego describe una solución general a dicho problema, que puede utilizarse en muchos contextos distintos. Para los patrones de diseño software, la solución consta normalmente de una descripción de un pequeño conjunto de clases junto con sus interacciones.

Los patrones de diseño nos ayudan en nuestra tarea de dos maneras distintas. En primer lugar, documentan una serie de buenas soluciones a los problemas, de modo que esas soluciones puedan emplearse posteriormente en problemas similares. La reutilización, en este caso, no se produce en el nivel del código fuente, sino en el nivel de las estructuras de clases.

En segundo lugar, los patrones de diseño tienen nombres, con lo que sirven para establecer un vocabulario que ayuda a los diseñadores software a hablar acerca de sus diseños. Cuando una serie de diseñadores experimentados hablan acerca de la estructura de una aplicación, uno de ellos podría decir: "Creo que aquí deberíamos utilizar un Singleton". *Singleton* es el nombre de un patrón de diseño bien conocido, por lo que si ambos diseñadores están familiarizados con el patrón, el ser capaces de hablar sobre él a este nivel ahorra el explicar un montón de detalles. Por tanto, el lenguaje de patrones introducido por diversos patrones de diseño bien conocidos nos permite alcanzar otro nivel de abstracción, gracias al cual podemos manejar la complejidad en sistemas que cada vez son más complejos.

Los patrones de diseño software se popularizaron gracias a un libro publicado en 1995 que describe un conjunto de patrones, sus aplicaciones y sus ventajas.[5] Este libro sigue siendo hoy día una de las obras más importantes acerca de patrones de diseño. Aquí, no pretendemos presentar una panorámica completa de esos patrones, sino que nos vamos a limitar a exponer un pequeño número de patrones, para que el lector pueda valorar las ventajas de la utilización de patrones de diseño; después, dejaremos que el lector continúe con el estudio de estos patrones de diseño en otros libros.

13.7.1 Estructura de un patrón

Las descripciones de patrones suelen hacerse utilizando una plantilla que contiene una cierta información mínima. Una descripción de un patrón no contiene solo información acerca de una estructura de clases, sino que también incluye una descripción del problema o problemas que este patrón resuelve y de las ventajas y desventajas del uso del patrón.

Una descripción de un patrón incluye al menos lo siguiente:

- Un **nombre** que puede utilizarse para hablar convenientemente del patrón.
- Una descripción del **problema** que el patrón resuelve (a menudo dividida en secciones como *intención*, *motivación* y *aplicabilidad*).
- Una descripción de la **solución** (que a menudo enumera la *estructura*, *participantes* y *colaboraciones*).
- Las **consecuencias** de utilizar el patrón, incluyendo los resultados y los compromisos.

---

[5] *Design Patterns: Elements of Reusable Object-Oriented Software* de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, Addison-Wesley, 1995.

En la siguiente sección, explicaremos brevemente algunos de los patrones utilizados más comúnmente.

13.7.2 Decorator

El patrón *Decorator* trata con el problema de añadir funcionalidad a un objeto existente. Asumimos que lo que se desea es disponer de un objeto que responda a las mismas llamadas a métodos (es decir, que tenga la misma interfaz) que el objeto existente, pero que tenga un comportamiento adicional o modificado. Puede que también queramos aumentar la interfaz existente.

Una forma en que esto puede hacerse es recurriendo a la herencia. Una subclase puede sustituir la implementación de algunos métodos y añadir otros métodos adicionales. Pero utilizar la herencia es una solución estática: una vez creados, los objetos no pueden modificar su comportamiento.

Una solución más dinámica es el uso de un objeto Decorator. Decorator es un objeto que encierra un objeto existente y que puede utilizarse en lugar del original (normalmente, implementa la misma interfaz). Los clientes pueden comunicarse con el Decorator en lugar de directamente con el original (sin tener por qué saber que se ha producido una sustitución). Decorator pasa las llamadas a métodos al objeto encerrado en él, pero también puede realizar acciones adicionales. Podemos encontrar un ejemplo en la librería de entrada/salida de Java. Allí, se utiliza un `BufferedReader` como Decorator para un `Reader` (Figura 13.2). El `BufferedReader` implementa la misma interfaz y puede emplearse en lugar de un `Reader` que no tiene mecanismo de buffer, pero añade funcionalidad adicional al comportamiento básico de un `Reader`. A diferencia de cuando se usa la herencia, los objetos decoradores pueden añadirse a objetos existentes.

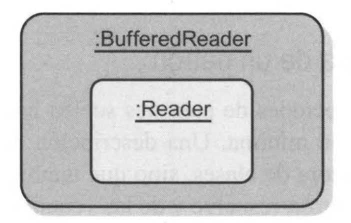


Figura 13.2
Estructura del patrón decorador.

13.7.3 Singleton

Una situación común en muchos programas es tener un objeto del cual solo deba existir una única instancia. Por ejemplo, en nuestro juego *world-of-zuul*, solo queríamos un único analizador sintáctico. Si escribimos un entorno de desarrollo software, nos encontraremos asimismo con que lo normal sea disponer de un único compilador o un único depurador.

El patrón *Singleton* garantiza que solo se cree una instancia de una clase y proporciona un acceso unificado a la misma. En Java, puede definirse un Singleton haciendo que el constructor sea privado. Esto garantiza que no pueda ser invocado desde fuera de la clase, por lo que las clases cliente no podrán crear nuevas instancias. Entonces, podemos crear código en la propia clase Singleton para crear esa única instancia y proporcionar acceso a la misma (El Código 13.1 ilustra esta técnica para una clase `Parser`).

Código 13.1
El patrón Singleton.

```
public class Parser
{
    private static Parser instance = new Parser();

    public static Parser getInstance()
    {
        return instance;
    }

    private Parser()
    {
        ...
    }
}
```

En este patrón:

- El constructor es privado, de modo que las instancias solo pueden ser creadas por la propia clase. Esto tiene que hacerse en una parte estática de la clase (inicializaciones de campos estáticos o métodos estáticos), porque de otro modo no existirá ninguna instancia.
- Se define un campo estático privado y se inicializa con la (única) instancia del analizador sintáctico.
- Se define un método estático `getInstance`, que proporciona acceso a esa única instancia.

Los clientes del Singleton pueden ahora utilizar ese método estático para obtener acceso al objeto analizador sintáctico:

```
Parser parser = Parser.getInstance();
```

13.7.4 Método Factoría

El patrón *método Factoría* proporciona una interfaz para crear objetos, pero deja que las subclases decidan qué clase específica de objeto se crea. Normalmente, el cliente espera una superclase o una interfaz del tipo dinámico correspondiente al objeto real y el método factoría proporciona especializaciones.

Los iteradores de las colecciones son un ejemplo de esta técnica. Si tenemos una variable de tipo `Collection`, podemos pedirla un objeto `Iterator` (utilizando el método `iterator`) y luego funcionar con ese iterador (Código 13.2). El método `iterator` de este ejemplo sería el método Factoría.

Código 13.2
Un uso de un método factoría.

```
public void process(Collection<Type> coll)
{
    Iterator<Type> it = coll.iterator();
    ...
}
```

Desde el punto de vista del cliente (en el código mostrado en el Código 13.2), estamos tratando con objetos de tipo `Collection` e `Iterator`. Pero en realidad, el tipo (dinámico) de

la colección puede ser ArrayList, en cuyo caso, el método iterator devuelve un objeto de tipo ArrayListIterator. O puede ser un HashSet, con lo que iterator devolverá un HashSetIterator. El método factoría se especializa en las subclases para devolver instancias especializadas de tipo de retorno "oficial".

Podemos hacer un buen uso de este patrón en nuestra simulación *foxes-and-rabbits*, para desacoplar la clase Simulator de las clases de animales específicas. (Recuerde: en nuestra versión, Simulator estaba acoplada con las clases Fox y Rabbit, porque crea las instancias iniciales.) En lugar de ello, podemos introducir una interfaz ActorFactory y una serie de clases que implementen esta interfaz para cada actor (por ejemplo, FoxFactory y RabbitFactory). El Simulator almacenaría simplemente una colección de objetos ActorFactory, y pediría a cada uno de ellos que generara una serie de actores. Cada factoría produciría, por supuesto, un tipo de actor diferente, pero el Simulator hablaría con ellos a través de la interfaz ActorFactory.

13.7.5 Observador

En las exposiciones de varios de los proyectos del libro, hemos tratado de separar el modelo interno de la aplicación de la manera en que esta se presenta en pantalla (la vista). El patrón *Observador* proporciona una forma de conseguir esa separación modelo/vista.

En términos más generales, el patrón Observador define una relación uno-a-muchos, de modo que cuando un objeto cambie su estado, muchos otros pueden ser notificados. Esto se consigue con un grado muy bajo de acoplamiento entre los observadores y los objetos observados.

Podemos ver con esto que el patrón Observador no solo soporta una vista desacoplada del modelo, sino que también permite que haya múltiples vistas distintas (bien como alternativas o bien simultáneamente). Como ejemplo, podemos utilizar de nuevo nuestra simulación *foxes-and-rabbits*.

En la simulación, presentábamos las poblaciones de animales en pantalla usando una cuadrícula animada bidimensional, pero existen otras posibilidades. Podríamos haber decidido mostrar la población con una gráfica del número de animales, junto con una línea temporal, o bien podríamos haberla presentado como un gráfico de barras animado (Figura 13.3). Podríamos incluso desear ver todas las representaciones al mismo tiempo.

Para el patrón Observador, utilizamos dos tipos: Observable y Observer.[6] La entidad observable (el objeto Field en nuestra simulación) amplía la clase Observable, y el observador (SimulatorView) implementa la interfaz Observer (Figura 13.4).

La clase Observable proporciona métodos para que los observadores se asocien ellos mismos a la entidad observada. Esto garantiza que se invoque el método update de los observadores cada vez que la entidad observada (en este caso, el hábitat) invoque su método heredado notify. Los observadores reales (los visualizadores) pueden entonces obtener un estado nuevo y actualizado del hábitat y redibujar el resultado de la simulación.

El patrón Observador también puede utilizarse para problemas distintos de los de la separación modelo/vista. Siempre puede aplicarse cuando el estado de uno o más objetos dependa del estado de otro objeto.

[6] En el paquete java.util de Java, Observer es en realidad una interfaz con un único método, update.

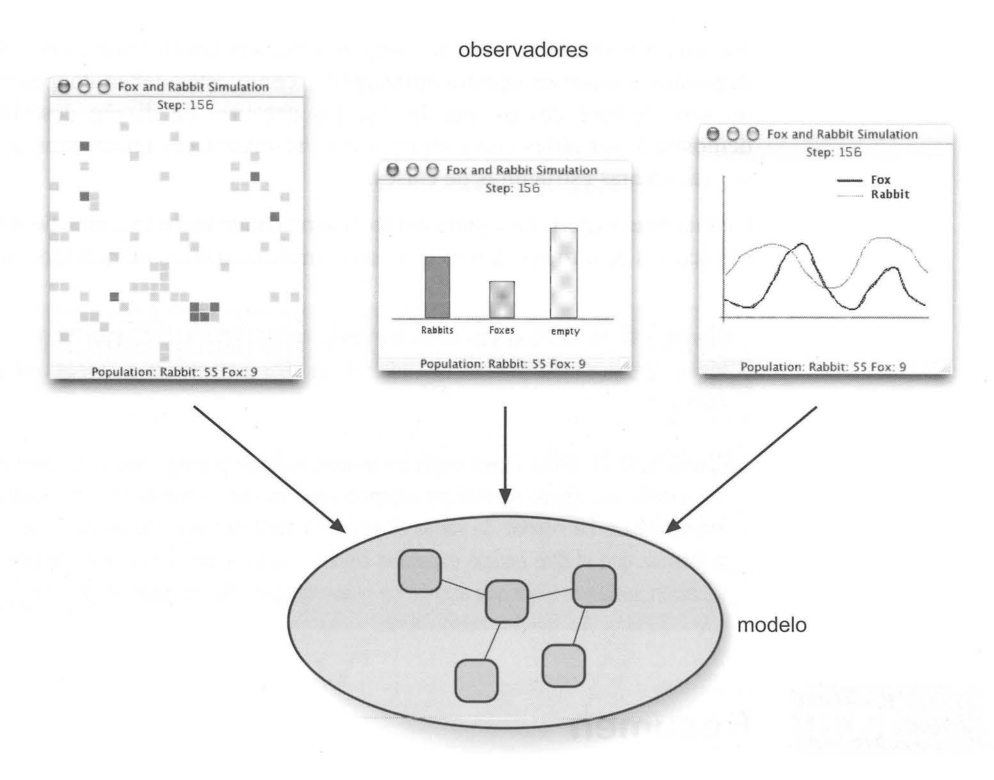


Figura 13.3
Múltiples vistas de un mismo sujeto.

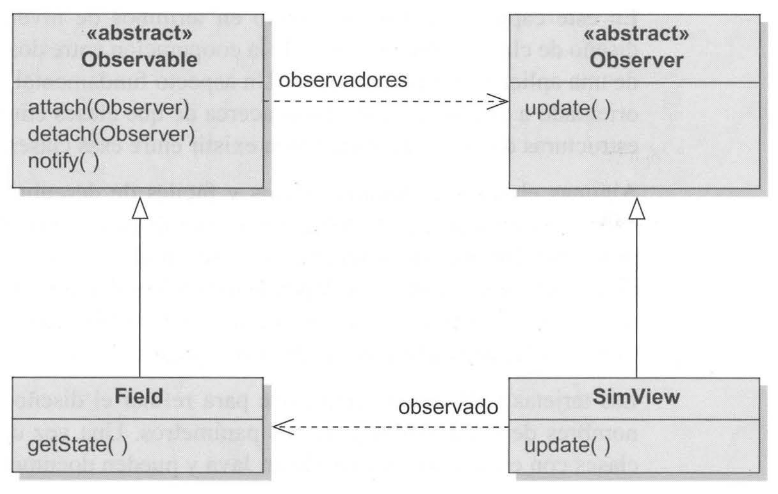


Figura 13.4
Estructura del patrón Observador.

13.7.6 Resumen de patrones

Analizar detalladamente los patrones de diseño y sus aplicaciones cae fuera del alcance de este libro. Aquí, hemos presentado únicamente una breve idea de lo que son los patrones de diseño y hemos proporcionado una descripción informal de algunos de los patrones más comunes.

Esperamos, sin embargo, que estas explicaciones le sirvan para ver a dónde dirigirse desde aquí. Una vez que se comprende cómo crear buenas implementaciones de clases individuales

con una funcionalidad bien definida, podemos concentrarnos en decidir en qué tipos de clases deberíamos tener en nuestra aplicación y cómo estas deben cooperar. Las buenas soluciones no son siempre obvias, por lo que los patrones de diseño describen estructuras que han demostrado ser útiles una y otra vez para resolver tipos recurrentes de problemas. Nos ayudan a crear buenas estructuras de clases.

Cuanta más experiencia gane como desarrollador software, más tiempo invertirá en pensar en estructuras de alto nivel, más que en la implementación de métodos individuales.

Ejercicio 13.16 Otros tres patrones comúnmente utilizados son el patrón *State*, el patrón *Strategy* y el patrón *Visitor*. Localice descripciones de cada uno de ellos e identifique al menos una aplicación de ejemplo para cada uno.

Ejercicio 13.17 En una fase tardía del desarrollo de un proyecto nos encontramos con que dos equipos que han estado trabajando de manera independiente en dos partes de una aplicación han implementado clases incompatibles. La interfaz de varias clases implementadas por uno de los equipos es ligeramente distinta de la interfaz que el otro equipo esperaba utilizar. Explique cómo el patrón *Adapter* puede ayudarnos en esta situación, evitando tener que escribir de nuevo ninguna de las clases existentes.

13.8 Resumen

En este capítulo, hemos ascendido en términos de niveles de abstracción, alejándonos del diseño de clases individuales (o de la cooperación entre dos clases) y centrándonos en el diseño de una aplicación en su conjunto. Un aspecto fundamental en el diseño de un sistema software orientado a objetos es la decisión acerca de qué clases emplear para su implementación y qué estructuras de comunicación deben existir entre esas clases.

Algunas clases son bastante obvias y fáciles de descubrir. Hemos utilizado como punto de partida un método de identificación de nombres y verbos dentro de la descripción textual del problema. Después de descubrir las clases, podemos utilizar tarjetas CRC y reproducciones de escenarios para diseñar las dependencias y los detalles de comunicación entre las clases, así como para concretar los detalles de las responsabilidades de cada clase. Para los diseñadores menos experimentados, suele servir de ayuda reproducir los escenarios en grupo.

Las tarjetas CRC puede emplearse para refinar el diseño hasta el nivel de definición de los nombres de los métodos y de sus parámetros. Una vez conseguido esto, pueden codificarse clases con esqueletos de método en Java y pueden documentarse las interfaces de las clases.

Seguir un proceso organizado como este tiene varias ventajas. Garantiza que los problemas potenciales relacionados con las primeras ideas de diseño se descubran antes de invertir mucho tiempo en su implementación. También permite a los programadores trabajar de manera independiente en la implementación de varias clases, sin tener que esperar a que esté terminada la implementación de una clase antes de poder comenzar con la implementación de otra.

Las estructuras de clases flexibles y ampliables no son siempre fáciles de diseñar. Se usan patrones de diseño para documentar estructuras generalmente adecuadas que han demostrado ser útiles en la implementación de diferentes tipos de problemas. A través del estudio de los

patrones de diseño, un ingeniero software puede aprender un montón acerca de lo que son buenas estructuras de aplicación y mejorar sus capacidades de diseño de aplicaciones.

Cuanto mayor sea la complejidad de un problema, más importante será disponer de una buena estructura de la aplicación. Cuanto más experimentado sea una ingeniero software, más tiempo invertirá en diseñar estructuras de aplicación que en solo escribir código.

Términos introducidos en el capítulo

análisis y diseño, métodos de los verbos/nombres, tarjeta CRC, escenario, caso de uso, esqueleto de método, patrón de diseño

Resumen de conceptos

- **verbos/nombres** Las clases de un sistema se corresponden aproximadamente con los nombres existentes en la descripción del sistema. Los métodos se corresponden con los verbos.
- **escenarios** Pueden utilizarse escenarios (también conocidos como "casos de uso") para comprender las interacciones dentro de un sistema.
- **prototipado** El prototipado es la construcción de un sistema parcialmente funcional, en el que algunas funciones de la aplicación están simuladas. Sirve para tratar de comprender en una fase temprana del proceso de desarrollo cómo funcionará el sistema.
- **patrón de diseño** Un patrón de diseño es una descripción de un problema común de programación, así como una descripción de un pequeño conjunto de clases (junto con su estructura de interacción) que ayuda a resolver el problema.

Ejercicio 13.18 Suponga que tiene un sistema de gestión escolar para una universidad. En él, hay una clase denominada `Database` (una clase bastante fundamental) que almacena objetos de tipo `Student`. Cada estudiante tiene una dirección que se almacena en un objeto `Address` (es decir, cada objeto `Student` tiene una referencia a un objeto `Address`).

Ahora, desde la clase `Database` necesitamos acceder a la calle, ciudad y código postal del estudiante. La clase `Address` dispone de métodos selectores para esto. Para el diseño de la clase `Student`, tendremos ahora dos opciones:

O bien implementamos métodos `getStreet`, `getCity` y `getZipCode` en la clase `Student` –que se limitarán a pasar la llamada a método al objeto `Address` y devolver el resultado que se les entregue– o bien se implementa un método `getAddress` en `Student` que devuelva el objeto `Address` completo a la instancia de `Database` y permita al objeto `Database` invocar directamente los métodos del objeto `Address`.

¿Cuál de estas alternativas es mejor? ¿Por qué? Haga un diagrama de clases para ambas soluciones y enumere los pros y los contras de cada una.

CAPÍTULO

14

Un caso de estudio

Principales conceptos explicados en el capítulo:

- desarrollo de aplicaciones completas

Estructuras Java explicadas en este capítulo:

(En este capítulo no se presenta ninguna nueva estructura Java.)

En este capítulo, vamos a unir o integrar muchos de los principios de orientación a objetos que hemos presentado a lo largo del libro, introduciendo un caso de estudio ampliado. Analizaremos ese estudio desde la exposición inicial del problema hasta las pruebas, pasando por el descubrimiento de clases, el diseño y un proceso iterativo de implementación y pruebas. A diferencia de los capítulos anteriores no es nuestra intención introducir aquí ningún nuevo concepto de importancia. Más bien, lo que intentamos es reforzar los temas que se han cubierto en la segunda parte del libro, como la herencia, las técnicas de abstracción, el tratamiento de errores y el diseño de aplicaciones.

14.1 El caso de estudio

El caso de estudio que emplearemos consiste en el desarrollo del modelo para una compañía de taxis. La empresa está considerando expandir sus servicios a una nueva parte de la ciudad, y trabaja tanto con taxis como con transportes lanzadera. Los taxis dejan a los pasajeros en el destino deseado antes de admitir nuevos pasajeros, mientras que las lanzaderas pueden recoger a varios pasajeros en distintas ubicaciones para hacer un mismo viaje, llevándoles a lugares similares (por ejemplo, recogiendo varios huéspedes en diferentes hoteles y llevándoles a diferentes terminales del aeropuerto). Basándose en las estimaciones del número de clientes potenciales en el nuevo área, la empresa quiere saber si una expansión les resultaría rentable y cuántos vehículos necesitaría para poder operar de manera efectiva.

14.1.1 Descripción del problema

El siguiente párrafo es una descripción informal de los procedimientos de operación de la compañía de taxis, descripción que se ha elaborado después de varias reuniones con los responsables de la empresa.

La compañía opera con taxis individuales y con lanzaderas. Los taxis se utilizan para transportar a una persona (o a un pequeño grupo) de un lugar a otro. Las lanzaderas se emplean para recoger personas en distintas ubicaciones y llevarlas a varios destinos. Cuando la empresa recibe la llamada de una persona, de un hotel, de un establecimiento recreativo o de una agencia de viajes, trata de planificar un vehículo para hacer el servicio. Si no dispone de vehículos libres, no tienen ningún tipo de sistema de espera ni de gestión de colas. Cuando un vehículo llega al lugar donde tiene que recoger a un cliente, el conductor notifica a la compañía. De forma similar, cuando se deja al pasajero en su destino, el conductor también lo notifica a la compañía.

Como hemos sugerido en el Capítulo 10, uno de los propósitos comunes del modelo consiste en ayudarnos a aprender cosas acerca de la situación que se está modelando. Es útil identificar en una etapa temprana lo que queremos saber, porque los objetivos resultantes pueden muy bien influir en el diseño que generemos. Por ejemplo, si tratamos de responder cuestiones acerca de la rentabilidad de operar una compañía de taxis en esta área, entonces debemos garantizar que el modelo nos proporcione información que nos ayude a evaluar la rentabilidad. Por tanto, dos cuestiones que debemos considerar son: cuántas veces se pierden clientes potenciales porque no hay ningún vehículo disponible para recogerlos y, en el extremo opuesto, cuánto tiempo permanecen inactivos los taxis por falta de pasajeros. Estas cuestiones no aparecen reflejadas en la descripción básica de cómo opera normalmente la empresa de taxis, pero representan escenarios con los que habrá que jugar cuando realicemos el diseño.

Por tanto, podríamos añadir el siguiente párrafo a la descripción:

El sistema almacena detalles acerca de las solicitudes de pasajeros que no pueden ser satisfechas. También proporciona detalles de cuánto tiempo invierten los vehículos en cada una de las siguientes actividades: transportando pasajeros, yendo hasta los lugares de recogida y permaneciendo inactivos.

Sin embargo, a medida que desarrollemos nuestro modelo nos centraremos simplemente en la descripción original de los procedimientos de operación de la empresa, dejando las características adicionales como ejercicios para el lector.

Ejercicio 14.1 ¿Hay algún dato adicional que cree que podría ser útil recopilar a partir del modelo? En caso afirmativo, añada esos requisitos a las descripciones dadas anteriormente y utilícelos en sus ampliaciones del proyecto.

14.2 Análisis y diseño

Como se sugiere en el Capítulo 13, comenzaremos tratando de identificar las clases e interacciones en la descripción del sistema, utilizando el método de los nombres y los verbos.

14.2.1 Descubrimiento de las clases

En la descripción están presentes los siguientes nombres (versiones en singular): compañía, taxi, lanzadera, persona, ubicación, destino, hotel, establecimiento recreativo, agencia de viajes, vehículo, servicio, lugar de recogida, conductor y pasajero.

El primer punto que hay que observar es que sería un error pasar directamente de esta lista de nombres a un conjunto de clases. Las descripciones informales rara vez están escritas en una forma que sea adecuada para establecer tal tipo de correspondencia directa.

Un refinamiento comúnmente necesario en la lista de nombres consiste en identificar los *sinónimos*: diferentes palabras utilizadas para la misma entidad. Por ejemplo, "persona" y "servicio" son ambas sinónimos de "pasajero".

Un refinamiento adicional consiste en eliminar aquellas entidades que realmente no es necesario modelar en el sistema. Por ejemplo, la descripción identificaba diversas formas en las que la compañía de taxis podría ser contactada: personas, hoteles, establecimientos recreativos y agencias de viajes. ¿Realmente es necesario mantener estas distinciones? La respuesta dependerá de la información que queramos obtener del modelo. Es posible que deseemos organizar descuentos para hoteles que proporcionen un gran número de clientes o enviar material publicitario a los establecimientos recreativos que no los proporcionen. Si no hace falta este nivel de detalle, entonces podemos simplificar el modelo simplemente "inyectando" pasajeros en él, de acuerdo con algún patrón estadísticamente razonable.

Ejercicio 14.2 Considere simplificar el número de nombres asociados con los vehículos. ¿Son en este contexto sinónimos "vehículo" y "taxi"? ¿Necesitamos distinguir entre "lanzadera" y "taxi"? ¿Y qué pasa con el tipo de vehículo y el "conductor"? Justifique sus respuestas.

Ejercicio 14.3 ¿Es posible eliminar cualquiera de los siguientes nombres, como sinónimos en este contexto: ubicación, destino y lugar de recogida?

Ejercicio 14.4 Identifique los nombres de cualquier ampliación que haya añadido al sistema y lleve a cabo las simplificaciones necesarias.

14.2.2 Utilización de tarjetas CRC

La Figura 14.1 contiene un resumen de las asociaciones entre nombres y verbos con las que nos quedamos, después haber hecho algunas simplificaciones en la descripción original. A cada uno de los nombres debería ahora asignársele una tarjeta CRC que utilizaremos para identificar las correspondientes responsabilidades y colaboradores.

A partir de este resumen, está claro que "taxi" y "lanzadera" son distintas especializaciones de una clase de vehículo más general. La distinción principal entre una lanzadera y un taxi es que un taxi solo se preocupa de recoger y transportar a un único pasajero o a un grupo coherente, mientras que una lanzadera trata concurrentemente con múltiples pasajeros *independientes*. La relación entre estos dos vehículos sugiere una jerarquía de herencia, en la que "taxi" y "lanzadera" representen subtipos de vehículo.

Ejercicio 14.5 Cree tarjetas CRC físicas para los nombres/clases identificados en esta sección, para poder trabajar con los escenarios sugeridos en la descripción del proyecto.

Ejercicio 14.6 Haga lo mismo para sus propias ampliaciones, si es que quiere seguir trabajando con ellas en las siguientes etapas.

Figura 14.1
Asociaciones entre los nombres y verbos de la compañía de taxis.

Nombres	Verbos
compañía	*opera taxis y lanzaderas* *recibe una llamada* *programa un vehículo*
taxi	*transporta un pasajero*
lanzadera	*transporta uno o más pasajeros*
pasajero	
ubicación	
origen del pasajero	*llama a la compañía*
vehículo	*recoge a un pasajero* *lleva al lugar de recogida* *notifica a la compañía la llegada* *notifica a la compañía que ha dejado al pasajero*

14.2.3 Escenarios

La compañía de taxis no representa realmente una aplicación muy compleja. Veremos que buena parte de la interacción total en el sistema se puede explorar jugando con el escenario fundamental consistente en tratar de satisfacer la solicitud de un pasajero para ir de un lugar de la ciudad a otro. En la práctica, este único escenario se descompondrá en una serie de pasos que se llevarán a cabo en secuencia, desde la llamada inicial hasta el acto de dejar al pasajero en su destino.

- Hemos decidido que un origen de pasajeros creará todos los nuevos objetos pasajero para el sistema. Por tanto, una responsabilidad de `PassengerSource`, el origen de pasajeros, será *Crear un pasajero* y un colaborador será `Passenger`.
- El origen de pasajeros llama a la compañía de taxis para solicitar que se recoja a un pasajero. Anotamos `TaxiCompany` como colaborador de `PassengerSource` y añadimos como responsabilidad *Solicitar una recogida.* De forma correspondiente, añadimos a `TaxiCompany` la responsabilidad de *Recibir una solicitud de recogida.* Asociados con la solicitud estarán un pasajero y un lugar de recogida. Por tanto, `TaxiCompany` tendrá `Passenger` y `Location` (lugares) como colaboradores. Cuando llame a la compañía con la solicitud, el origen de pasajeros puede pasar el pasajero y el lugar de recogida como objetos separados. Sin embargo, es preferible asociar estrechamente el lugar de recogida con el pasajero. Por tanto, un colaborador de `Passenger` será `Location`, y una responsabilidad será *Proporcionar el lugar de recogida.*
- ¿Dónde se origina el lugar de recogida del pasajero? El lugar de recogida y el destino podrían decidirse en el momento de crear el objeto `Passenger`. Por tanto, añadimos a `PassengerSource` la responsabilidad de *Generar los lugares de recogida y de destino para un pasajero*, con `Location` como colaborador; y añadimos a `Passenger` las responsabilidades de *Recibir los lugares de recogida y destino* y de *Proporcionar el lugar de destino*.
- Al recibir una solicitud, la `TaxiCompany` tiene la responsabilidad de *Enviar un vehículo.* Esto sugiere que una responsabilidad adicional es *Almacenar una colección de vehículos*, con `Collection` y `Vehicle` como colaboradores. Puesto que la solicitud puede fallar (existe

la posibilidad de que no haya vehículos libres), habrá que devolver una indicación de éxito o de fallo al origen de pasajeros.

- No hay ninguna indicación de si la compañía quiere distinguir entre taxis y lanzaderas a la hora de realizar la planificación, por lo que no tenemos ninguna necesidad de tener en cuenta ese aspecto aquí. Sin embargo, un vehículo solo puede ser enviado si está libre. Esto significa que una responsabilidad de `Vehicle` será *Indicar si está libre.*
- Cuando se ha identificado un vehículo libre, hay que dirigirlo al lugar de recogida. `TaxiCompany` tiene la responsabilidad de *Dirigir el vehículo al lugar de recogida*, con la correspondiente responsabilidad en `Vehicle` de *Recibir lugar de recogida.* `Location` se añade como colaborador de `Vehicle`.
- Al recibir un lugar de recogida, el comportamiento de los taxis y lanzaderas puede diferir. Un taxi solo habrá estado libre si no se encontraba ya de camino a un lugar de recogida o de destino. Por tanto, la responsabilidad de `Taxi` es *Ir al lugar de recogida.* Por el contrario, una lanzadera tiene que tratar con varios pasajeros. Cuando recibe un lugar de recogida puede que tenga que elegir entre varios lugares alternativos posibles, para decidir a cuál se encamina a continuación. Por tanto, añadimos a `Shuttle` (la lanzadera) la responsabilidad de *Elegir el siguiente lugar de destino*, con un colaborador `Collection` para mantener el conjunto de posibles lugares de destino entre los que tendrá que elegir. El hecho de que un `Vehicle` se desplace de un lugar a otro sugiere que tiene la responsabilidad de *Mantener una posición actual.*
- Al llegar al lugar de recogida, un objeto `Vehicle` deberá *Notificar a la compañía que ha llegado al lugar de recogida*, con `TaxiCompany` como un colaborador; `TaxiCompany` deberá *Recibir la notificación de llegada al lugar de recogida.* En la vida real, un taxi se encuentra con sus pasajeros por primera vez cuando llega al lugar de recogida, así que es el punto natural para que el vehículo reciba a su siguiente pasajero. En el modelo, lo recibe de la compañía, que a su vez lo ha recibido del origen de pasajeros. Responsabilidad de `TaxiCompany`: *Pasar el pasajero al vehículo*; responsabilidad de `Vehicle`: *Recibir al pasajero*, con `Passenger` añadido como colaborador a `Vehicle`.
- El vehículo ahora solicita el destino del pasajero. Responsabilidad de `Vehicle`: *Solicitar el lugar de destino*; responsabilidad de `Passenger`: *Proporcionar el lugar de destino.* De nuevo, en este punto, el comportamiento de los taxis y las lanzaderas será diferente. Un `Taxi` simplemente se encargará de *Ir al destino del pasajero.* Una lanzadera se encargará de *Añadir un lugar a la colección de lugares de destino* y seleccionar el siguiente lugar al que tiene que ir.
- Al llegar al destino de un pasajero, un `Vehicle` tiene las responsabilidades de *Descargar al pasajero* y *Notificar a la compañía la llegada del pasajero a su destino.* La `TaxiCompany` deberá *Recibir la notificación de la llegada del pasajero a su destino.*

Los pasos que hemos esbozado representan la actividad fundamental de la compañía de taxis, repetidos una y otra vez a medida que nuevos pasajeros solicitan el servicio. Un punto importante que hay que resaltar, sin embargo, es que nuestro modelo software necesita poder reiniciar la secuencia para cada nuevo pasajero en cuanto se reciba cada nueva solicitud, aunque los servicios derivados de las solicitudes anteriores no se hayan completado todavía. En otras palabras, dentro de cada paso del programa podría haber un vehículo que todavía se esté dirigiendo al lugar de recogida, mientras que otro podría estar llegando al lugar de destino de un pasajero y un nuevo pasajero podría estar solicitando que le recojan.

Ejercicio 14.7 Revise la descripción del problema y el escenario que hemos analizado. ¿Hay algún otro escenario que tengamos que contemplar antes de pasar al diseño de clases? ¿Hemos cubierto adecuadamente lo que sucede, por ejemplo, si no hay ningún vehículo disponible en el momento de recibirse una solicitud? Complete el análisis de escenarios si piensa que todavía quedan cosas por hacer.

Ejercicio 14.8 ¿Cree que hemos descrito el escenario con el nivel correcto de detalle? Por ejemplo, ¿hemos incluido demasiado o demasiado poco detalle en lo que se refiere a las diferencias entre taxis y lanzaderas?

Ejercicio 14.9 ¿Cree que es necesario contemplar *cómo* se mueven los vehículos de un lugar a otro en esta etapa?

Ejercicio 14.10 ¿Cree que surgirá la necesidad de incluir clases adicionales a medida que desarrollemos la aplicación, clases que no tengan una referencia inmediata dentro de la descripción del problema? En caso afirmativo, ¿por qué cree que es así?

14.3 Diseño de clases

En esta sección, comenzaremos el proceso de pasar desde un diseño abstracto de alto nivel hecho en papel a un esbozo concreto de diseño dentro de un proyecto BlueJ.

14.3.1 Diseño de las interfaces de las clases

En el Capítulo 13, hemos sugerido que el siguiente paso consistía en crear un nuevo conjunto de tarjetas CRC a partir del primero, transformando las responsabilidades de cada clase en un conjunto de signaturas de métodos. Sin querer quitar énfasis a la importancia de dicho paso, vamos a dejárselo al lector como ejercicio y a pasar directamente a un esbozo de proyecto BlueJ que contenga esqueletos de clases y métodos. Esto nos proporcionará una idea aproximada de la complejidad del proyecto y nos dirá si nos hemos dejado algo crucial en los pasos que hemos dado hasta ahora.

Merece la pena resaltar que, en cada etapa del ciclo de vida del proyecto, deberíamos esperar encontrar errores o cabos sueltos en aquello que hemos hecho en las etapas anteriores. Esto no implica necesariamente que haya debilidades en nuestras técnicas o que nos falte capacidad. Esto más bien refleja el hecho de que el desarrollo de un proyecto es a menudo un proceso de descubrimiento. Solo explorando y probando las cosas llegamos a comprender del todo qué es lo que estamos intentando conseguir. ¡Así que el descubrir omisiones nos está diciendo en realidad algo positivo acerca del proceso que estamos siguiendo!

14.3.2 Colaboradores

Habiendo identificado las colaboraciones entre clases, un problema que tendrá que ser abordado a menudo es cómo obtiene cada objeto concreto las referencias a sus colaboradores. Suele haber tres formas distintas en las que esto puede suceder, y cada forma representa un patrón distinto de interacción entre objetos:

- Un colaborador se recibe como argumento de un constructor. Dicho colaborador será normalmente almacenado en uno de los campos del nuevo objeto, para que esté disponible

a lo largo de la vida del nuevo objeto. El colaborador puede ser compartido de esta forma con varios objetos diferentes. Ejemplo: un objeto `PassengerSource` recibe el objeto `TaxiCompany` a través de su constructor.

- Se recibe un colaborador como argumento de un método. La interacción con dicho colaborador es usualmente transitoria –simplemente mientras dura la ejecución del método–, aunque el objeto receptor puede decidir almacenar la referencia en uno de sus campos para una interacción a más largo plazo. Ejemplo: `TaxiCompany` recibe un colaborador `Passenger` a través de su método para gestionar una solicitud de recogida de pasajero.
- El objeto construye el colaborador por sí mismo. Ese colaborador será para el uso exclusivo del objeto que lo construye, a menos que se pase a otro objeto en alguna de las dos maneras anteriores. Si se construye en un método, la colaboración será normalmente a corto plazo, mientras dure el bloque dentro del cual ha sido construido. Sin embargo, si el colaborador está almacenado en un campo, entonces la colaboración es probable que dure todo el tiempo de vida del objeto creador. Ejemplo: `TaxiCompany` crea una colección para almacenar sus vehículos.

Ejercicio 14.11 A medida que analicemos en la siguiente sección el proyecto *taxi-company-outline*, preste especial atención a dónde se crean los objetos y cómo los objetos que están colaborando llegan a conocerse unos a otros. Trate de identificar al menos un ejemplo más de cada uno de los patrones que hemos descrito.

14.3.3 El esbozo de implementación

El proyecto *taxi-company-outline* contiene un esbozo de implementación de las clases, responsabilidades y colaboraciones que hemos descrito como parte del proceso de diseño. Le animamos a explorar el código fuente y a asociar las clases concretas con la descripción correspondiente en la Sección 14.2.3. El Código 14.1 muestra una esbozo de la clase `Vehicle` del proyecto.

Código 14.1
Un esbozo de la clase `Vehicle`.

```
/**
 * Capturar un esbozo de los datos de un vehículo.
 *
 * @author David J. Barnes y Michael Kölling
 * @version 2011.07.31
 */
public abstract class Vehicle
{
    private TaxiCompany company;
    // Lugar donde está el vehículo.
    private Location location;
    // Lugar al que se dirige el vehículo.
    private Location targetLocation;

    /**
     * Constructor de la clase Vehicle.
```

Código 14.1 (*continuación*) Un esbozo de la clase `Vehicle`.

```
     * @param company La compañía de taxis. No debe ser null.
     * @param location El punto de partida del vehículo.
     *                 No debe ser null.
     * @throws NullPointerException Si la compañía o el lugar de
     *                  partida son null.
     */
    public Vehicle(TaxiCompany company, Location location)
    {
        if(company == null) {
            throw new NullPointerException("company");
        }
        if(location == null) {
            throw new NullPointerException("location");
        }
        this.company = company;
        this.location = location;
        targetLocation = null;
    }

    /**
     * Notificar a la compañía de la llegada a un lugar
     * de recogida.
     */
    public void notifyPickupArrival()
    {
        company.arrivedAtPickup(this);
    }

    /**
     * Notificar a la compañía de la llegada al lugar
     * de destino de un pasajero.
     */
    public void notifyPassengerArrival(Passenger passenger)
    {
        company.arrivedAtDestination(this, passenger);
    }

    /**
     * Recibir un lugar de recogida.
     * Cómo se gestione esto dependerá del tipo de vehículo.
     * @param location El lugar de recogida.
     */
    public abstract void setPickupLocation(Location location);

    /**
     * Recibir un pasajero.
     * Cómo se gestione esto dependerá del tipo de vehículo.
     * @param passenger El pasajero.
     */
```

Código 14.1
(*continuación*)
Un esbozo de la clase Vehicle.

```
    public abstract void pickup(Passenger passenger);
    /**
     * Obtener el lugar de destino.
     * @return Si este vehículo está o no libre.
     */
    public abstract boolean isFree();

    /**
     * Descargar a todos los pasajeros cuyo destino sea
     * el lugar actual.
     */
    public abstract void offloadPassenger();

    /**
     * @return Lugar donde se encuentra actualmente este vehículo.
     */
    public Location getLocation()
    {
        return location;
    }

    /**
     * Establecer la ubicación actual.
     * @param location Cuál es esa ubicación. No debe ser null.
     * @throws NullPointerException Si la ubicación es null.
     */
    public void setLocation(Location location)
    {
        if(location != null) {
            this.location = location;
        }
        else {
            throw new NullPointerException();
        }
    }

    /**
     * @return Lugar al que se dirige actualmente este vehículo,
     *         o null si está inactivo.
     */
    public Location getTargetLocation()
    {
        return targetLocation;
    }

    /**
     * Establecer el lugar de destino requerido.
     * @param location A dónde ir. No debe ser null.
```

Código 14.1 (*continuación*) Un esbozo de la clase `Vehicle`.

```
     * @throws NullPointerException Si el lugar es null.
     */
    public void setTargetLocation(Location location)
    {
        if (location != null) {
            targetLocation = location;
        }
        else {
            throw new NullPointerException();
        }
    }

    /**
     * Borrar el lugar de destino.
     */
    public void clearTargetLocation()
    {
        targetLocation = null;
    }
}
```

El proceso de crear el esbozo de proyecto ha hecho que se nos planteen una serie de cuestiones. He aquí algunas de ellas:

- Siempre cabe esperar encontrar algunas diferencias entre el diseño y la implementación, debido a la diferente naturaleza de los lenguajes de implementación y de diseño. Por ejemplo, el análisis de los escenarios sugería que `PassengerSource` debería tener la responsabilidad de *Generar los lugares de recogida y de destino para un pasajero*, mientras que `Passenger` debería tener la responsabilidad de *Recibir los lugares de recogida y de destino*. En lugar de asignar estas responsabilidades a llamadas individuales a métodos, la implementación más natural en Java sería escribir algo como

```
new Passenger(new Location(...), new Location(...))
```

- Hemos hecho lo necesario para que nuestro esbozo de proyecto esté lo suficientemente completo como para poder compilarse sin problemas. Eso no siempre es necesario en esta etapa, pero sí que implica que el realizar un desarrollo incremental en la etapa siguiente nos resultará algo más fácil. Sin embargo, tiene a cambio la desventaja de hacer que sea potencialmente más difícil localizar los fragmentos de código que faltan, porque el compilador no nos señalará los cabos sueltos que nos hayamos podido dejar.

- Los elementos compartidos y los elementos diferenciadores de las clases `Vehicle`, `Taxi` y `Shuttle` solo comenzarán a tomar forma a medida que progresemos en su implementación. Por ejemplo, las diferentes formas en las que los taxis y las lanzaderas responden a una solicitud de recogida se ven reflejadas en el hecho de que `Vehicle` define `setPickupLocation` como un método abstracto, que tendrá implementaciones concretas distintas dentro de las subclases. Por otro lado, aunque los taxis y las lanzaderas tienen diferentes formas de decidir hacia dónde tienen que dirigirse, sí que pueden compartir el concepto de disponer de un único lugar de destino. Esto se ha implementado como un campo `targetLocation` en una superclase.

- En dos puntos del escenario, se espera que cada vehículo notifique a la compañía de taxis su llegada a un punto de recogida o de destino. Hay al menos dos maneras posibles de organizar esto en la implementación. La forma directa consiste en que el vehículo almacena una referencia a su compañía. Esto implica que habría una asociación explícita entre las dos clases en el diagrama de clases.

  Una alternativa sería utilizar el patrón *Observador* presentado en el Capítulo 13, con `Vehicle` ampliando la clase `Observable` y `TaxiCompany` implementando la interfaz `Observer`. El acoplamiento directo entre `Vehicle` y `TaxiCompany` se reduce, pero continuará habiendo un acoplamiento implícito y el proceso de notificaciones es algo más complejo de programar.

- Hasta este punto, no hemos hablado de cuántos pasajeros puede transportar una lanzadera. Presumiblemente, podrían existir lanzaderas de distintos tamaños. Este aspecto de la aplicación ha sido diferido para resolverlo posteriormente.

No hay ninguna regla absoluta que nos indique exactamente lo lejos que debemos ir con el esbozo de implementación en una aplicación concreta. El propósito del esbozo de implementación no es crear un proyecto completamente funcional, sino dejar constancia del diseño del esbozo de estructura de la aplicación (que ha sido desarrollado mediante las actividades con tarjetas CRC a las que antes hemos hecho referencia). A medida que revise las clases en el proyecto *taxi-company-outline*, puede que crea que hemos ido demasiado lejos en este caso, o puede, por el contrario, que piense que no hemos llegado lo suficientemente lejos. En el lado positivo, al intentar crear una versión que al menos se compila, hemos comprobado que nos veíamos forzados a pensar en la jerarquía de herencia de `Vehicle` con un cierto detalle: en particular, hemos tenido que ver qué métodos podían implementarse completamente en la superclase y cuáles era mejor definir como abstractos. En el lado negativo, existe el riesgo de tomar decisiones de implementación demasiado pronto: por ejemplo, comprometerse con tipos concretos de estructuras de datos que podrían perfectamente ser seleccionados más tarde o, como hemos hecho aquí, decidir rechazar el patrón *Observador* en favor de la solución más directa.

Ejercicio 14.12 Para cada una de las clases del proyecto, examine la interfaz de la clase y escriba una lista de pruebas JUnit que podrían utilizarse para probar la funcionalidad de la clase.

Ejercicio 14.13 El proyecto *taxi-company-outline* define una clase `Demo` para crear una pareja de objetos `PassengerSource` y `TaxiCompany`. Cree un objeto Demo y pruebe su método `pickupTest`. ¿Por qué el objeto `TaxiCompany` es incapaz de satisfacer una solicitud de recogida en esta fase?

Ejercicio 14.14 ¿Cree que deberíamos haber desarrollado más el código fuente en esta etapa, para permitir que al menos una solicitud de recogida sea satisfecha con éxito? En caso afirmativo, ¿cuánto más lejos habría llevado usted el desarrollo?

14.3.4 Pruebas

Habiendo comenzado con la implementación no deberíamos progresar mucho más antes de comenzar a considerar cómo vamos a probar la aplicación. No queremos cometer el error de desarrollar pruebas únicamente después de que la implementación se haya completado. Podemos incorporar algunas pruebas que irán evolucionando gradualmente a medida que lo

haga la implementación. Haga los siguientes ejercicios para ver qué tipo de cosas es posible probar en esta etapa tan temprana.

Ejercicio 14.15 El proyecto *taxi-company-outline-testing* incluye algunas pruebas JUnit iniciales muy sencillas. Experimente con ellas. Añada alguna prueba adicional que crea que es apropiada en esta etapa del desarrollo, para formar la base de un conjunto de pruebas que se utilizarán durante el desarrollo futuro. ¿Piensa que importa que fallen las pruebas que creemos en esta etapa?

Ejercicio 14.16 La clase `Location` actualmente no contiene ningún campo ni método. ¿Cómo cree que el desarrollo posterior de esta clase afectará a las clases de prueba existentes?

14.3.5 Algunos problemas pendientes

Uno de los principales problemas que no hemos intentado abordar todavía es cómo organizar la secuencia de las diversas actividades: solicitudes de pasajeros, movimientos de vehículos, etc. Otra cuestión es que no hemos dado a los lugares una forma detallada concreta, por lo que el movimiento no tiene ningún efecto. A medida que vayamos desarrollando más la aplicación, irán emergiendo las soluciones para estos y otros problemas.

14.4 Desarrollo iterativo

Obviamente, todavía nos queda un largo camino que recorrer desde el esbozo de implementación desarrollado en el proyecto *taxi-company-outline* hasta la versión final. Sin embargo, en lugar de dejarnos abrumar por la magnitud de la tarea global, podemos hacer que las cosas sean más manejables identificando algunos pasos discretos que podemos dar hacia nuestro objetivo final y acometiendo un proceso de desarrollo iterativo.

14.4.1 Pasos del desarrollo

Planificar algunos pasos de desarrollo nos ayuda a considerar cómo podríamos descomponer un único problema de gran tamaño en varios problemas más pequeños. Tomados individualmente, es probable que estos problemas más pequeños sean a la vez menos complejos y más manejables que el problema global de mayor envergadura; al mismo tiempo, esos problemas se combinan para formar un todo coherente. A medida que tratemos de resolver los problemas de menor tamaño, podemos encontrarnos con que también necesitamos descomponer algunos de ellos. Además, podríamos ver que algunas de nuestras suposiciones originales eran erróneas o que nuestro diseño resulta inadecuado de alguna manera. Este proceso de descubrimiento, al combinarse con un enfoque de desarrollo iterativo, implica que iremos obteniendo datos muy valiosos sobre lo adecuado de nuestro diseño en lo relativo a las decisiones que hayamos tomado; y además, esos datos los obtendremos en una etapa lo suficientemente temprana como para poder incorporarlos dentro de un proceso flexible y de carácter evolutivo.

Considerar los pasos en los que podemos descomponer el problema global tiene la ventaja adicional de ayudarnos a identificar algunas de las formas en que se interconectan las distintas partes de la aplicación. En un proyecto grande, este proceso nos ayuda a identificar las

interfaces entre componentes. Identificar los pasos también ayuda a planificar los tiempos del proceso de desarrollo.

Es importante que cada paso de un desarrollo iterativo represente un punto claramente identificable de la evolución de la aplicación, mientras esta va progresando para satisfacer los requisitos globales. En particular, necesitamos poder determinar cuándo se ha completado cada paso. La finalización de cada paso debería quedar marcada por la realización satisfactoria de un conjunto de pruebas y por una revisión de los logros alcanzados durante ese paso, con el fin de poder incorporar en los pasos siguientes las lecciones aprendidas.

He aquí una serie posible de pasos de desarrollo para la aplicación de la compañía de taxis:

- Permitir que un único taxi pueda recoger a un único pasajero y llevarlo a su destino.
- Proporcionar suficientes taxis como para permitir recoger a varios pasajeros independientes y llevarlos a sus destinos de forma concurrente.
- Permitir que una única lanzadera recoja a un único pasajero y lo lleve a su destino.
- Garantizar que se registren los detalles de los pasajeros para los que no haya ningún vehículo libre.
- Permitir que una misma lanzadera recoja a varios pasajeros y los lleve a sus destinos de forma concurrente.
- Proporcionar una GUI para visualizar las actividades de todos los pasajeros y vehículos activos dentro de la simulación.
- Asegurarse de que los taxis y las lanzaderas sean capaces de operar de forma concurrente.
- Proporcionar toda la funcionalidad restante, incluyendo datos estadísticos completos.

No vamos a analizar en detalle la implementación de todos estos pasos, pero sí que vamos a completar la aplicación hasta un punto en el que el lector debería poder añadir por sí mismo la funcionalidad restante.

Ejercicio 14.17 Evalúe de manera crítica la lista de pasos que hemos esbozado, teniendo presentes las siguientes cuestiones: ¿Cree que el orden es apropiado? ¿Cree que el nivel de complejidad de cada paso es excesivo, escaso o simplemente correcto? ¿Cree que falta algún paso? Modifique la lista de la forma que crea conveniente para adaptarla a su propia visión del proyecto.

Ejercicio 14.18 ¿Son suficientemente obvios los criterios de finalización (pruebas de finalización) de cada etapa? En caso afirmativo, documente algunas de las pruebas que podrían realizarse al final de cada paso.

14.4.2 Una primera etapa

Para la primera etapa, queremos poder crear un único pasajero, hacer que sea recogido por un único taxi y lograr que sea llevado hasta su destino. Esto significa que tendremos que trabajar con varias clases distintas: por supuesto, `Location`, `Taxi` y `TaxiCompany`, y posiblemente otras. Además, tendremos que hacer que vaya transcurriendo un tiempo simulado a medida que el taxi se desplaza por la ciudad. Esto sugiere que quizá podamos reutilizar algunas de las ideas relativas a los actores que vimos en el Capítulo 10.

El proyecto *taxi-company-stage-one* contiene una implementación de los requisitos de esta primera etapa. Se han desarrollado las clases hasta el punto en que un taxi puede recoger y llevar a un pasajero hasta su destino. El método `run` de la clase `Demo` se encarga de ejecutar este escenario. Sin embargo, lo más importante en esta etapa son en realidad las clases de prueba –`LocationTest`, `PassengerTest`, `PassengerSourceTest` y `TaxiTest`– de las que hablaremos en la Sección 14.4.3.

En lugar de analizar este proyecto en detalle, vamos a describir simplemente algunos de los problemas que surgieron al desarrollarlo partiendo del esbozo de implementación anterior. Debería tratar de complementar este análisis con una lectura en profundidad del código fuente.

Los objetivos de la primera etapa se establecieron deliberadamente como bastante modestos, pero de manera que siguieran siendo lo suficientemente relevantes para la actividad fundamental de la aplicación –recoger y llevar a los pasajeros a sus destinos. Teníamos buenas razones para hacer esto. Al establecer un objetivo modesto, parecía que la tarea se podía completar en un tiempo razonable. Al establecer un objetivo más ambicioso, la tarea nos lleva además, claramente, hacia nuestro objetivo final, consistente en completar el proyecto global. Dichos factores ayudan a mantener un alto grado de motivación.

Hemos tomado prestado el concepto de actores del proyecto *foxes-and-rabbits* del Capítulo 10. Para esta etapa, solo necesitamos que sean actores los taxis, a través de su superclase `Vehicle`. En cada etapa, un taxi se mueve hacia un lugar de destino o permanece inactivo (Código 14.2). Aunque no teníamos que registrar ninguna estadística en esta etapa, resultaba simple y conveniente hacer que los vehículos registraran el número de pasos durante el cual están inactivos. Esto nos permite anticipar parte del trabajo de una de las etapas posteriores.

Código 14.2
La clase `Taxi` como un actor.

```
/**
 * Un taxi es capaz de transportar a un único pasajero.
 *
 * @author David J. Barnes y Michael Kölling
 * @version 2011.07.31
 */
public class Taxi extends Vehicle
{
    private Passenger passenger;

    /**
     * Constructor de los objetos de la clase Taxi
     * @param company La compañía de taxis. No debe ser null.
     * @param location El punto de partida del vehículo.
     *                 No debe ser null.
     * @throws NullPointerException Si la compañía o el lugar de
     *                              partida son null.
     */
    public Taxi(TaxiCompany company, Location location)
    {
        super(company, location);
    }
```

Código 14.2 (*continuación*) La clase `Taxi` como un actor.

```
    /**
     * Llevar a cabo las acciones de un taxi.
     */
    public void act()
    {
        Location target = getTargetLocation();
        if(target != null) {
            // Find where to move to next.
            Location next = getLocation().nextLocation(target);
            setLocation(next);
            if(next.equals(target)) {
                if(passenger != null) {
                    notifyPassengerArrival(passenger);
                    offloadPassenger();
                }
                else {
                    notifyPickupArrival();
                }
            }
        }
        else {
            incrementIdleCount();
        }
    }

    /**
     * ¿Está libre el taxi?
     * @return Si el taxi está libre o no.
     */
    public boolean isFree()
    {
        return getTargetLocation() == null && passenger == null;
    }

    /**
     * Recibir un lugar de recogida. Este será el lugar
     * de destino.
     * @location El lugar de recogida.
     */
    public void setPickupLocation(Location location)
    {
        setTargetLocation(location);
    }

    /**
     * Recibir un pasajero.
     * Establecer como destino el lugar de destino del pasajero.
```

Código 14.2 (*continuación*) La clase `Taxi` como un actor.

```
    * @param passenger El pasajero.
    */
   public void pickup(Passenger passenger)
   {
       this.passenger = passenger;
       setTargetLocation(passenger.getDestination());
   }
   /**
    * Descargar al pasajero.
    */
   public void offloadPassenger()
   {
       passenger = null;
       clearTargetLocation();
   }

   /**
    * Devolver detalles del taxi, como por ejemplo el lugar
    * en el que se encuentra.
    * @return Una representación del taxi en forma de cadena.
    */
   public String toString()
   {
       return "Taxi at " + getLocation();
   }
}
```

La necesidad de modelar el movimiento nos ha obligado a implementar la clase `Location` de forma más completa que en el esbozo. Aparentemente, debería ser un contenedor relativamente simple para una posición bidimensional dentro de cuadrícula rectangular. Sin embargo, en la práctica, también tiene que proporcionar tanto una prueba de coincidencia de dos lugares (`equals`), como alguna manera de que un vehículo pueda determinar a dónde debe ir a continuación, basándose en su posición actual y su destino (`nextLocation`). En esta etapa, no hemos puesto ningún límite relativo al área de la cuadrícula (salvo que los valores de coordenadas deben ser positivos), pero esto hace que surja la necesidad de incluir en una etapa posterior algo que indique las fronteras del área en la que opera la compañía.

Uno de los problemas fundamentales que hubo que abordar fue cómo gestionar la asociación entre un pasajero y un vehículo, entre una solicitud de recogida y el punto de llegada del vehículo. Aunque solo se nos pedía gestionar un único taxi y un único pasajero, hemos tratado de tener en mente que al final podría haber múltiples solicitudes de recogida pendientes en cualquier momento concreto. En la Sección 14.2.3, habíamos decidido que un vehículo debería recibir su pasajero cuando notificara a la compañía que había llegado al lugar de recogida. Por tanto, cuando se recibe una notificación, la compañía debe poder determinar qué pasajero se ha asignado a ese vehículo. La solución que elegimos fue que la compañía almacenara una pareja *vehículo:pasajero* dentro de un mapa. Cuando el vehículo notifica a la compañía que ha llegado, la compañía le pasa el pasajero correspondiente. Sin embargo, hay varias razones

por las que esta solución no es perfecta, y exploraremos esta cuestión con más detalle en los siguientes ejercicios.

Una situación de error que hemos contemplado es que podría no encontrarse el pasajero cuando un vehículo llegue a un punto de recogida. Esto sería el resultado de un error de programación, así que hemos definido la clase de excepción no comprobada `MissingPassengerException`.

Como solo se nos exigía un único pasajero en esta etapa, hemos diferido el desarrollo de la clase `PassengerSource` a una etapa posterior. En lugar de usar un origen de pasajeros, hemos creado directamente los pasajeros en las clases `Demo` y `Test`.

Ejercicio 14.19 Si todavía no lo ha hecho, examine la implementación del proyecto *taxi-company-stage-one*. Asegúrese de comprender cómo se efectúa el movimiento del taxi a través de su método `act`.

Ejercicio 14.20 ¿Cree que el objeto `TaxiCompany` debería mantener listas separadas de los vehículos que están libres y de los que no, para mejorar la eficiencia de sus planificaciones? ¿En qué puntos pasaría un vehículo de una lista a otra?

Ejercicio 14.21 La siguiente etapa planificada de la implementación consiste en proporcionar múltiples taxis para transportar varios pasajeros de manera concurrente. Revise la clase `TaxiCompany` con este objetivo en mente. ¿Cree que ya soporta esta funcionalidad? En caso negativo, ¿qué cambios son necesarios?

Ejercicio 14.22 Revise la forma en la que se almacenan las asociaciones *vehículo:pasajero* en el mapa de asignaciones de `TaxiCompany`. ¿Detecta alguna debilidad en este enfoque? ¿Permite que se recoja a más de un pasajero en el mismo lugar? ¿Podría un vehículo necesitar alguna vez que se registraran múltiples asociaciones para él?

Ejercicio 14.23 Si ve algún problema con la forma en que actualmente se almacenan las asociaciones *vehículo:pasajero*, ¿cree que serviría de ayuda crear una identificación unívoca para cada asociación, como por ejemplo un "número de reserva"? En caso afirmativo, ¿Sería necesario modificar cualquiera de las signaturas de métodos existentes en la jerarquía de `Vehicle`? Implemente una versión mejorada que soporte los requisitos de todos los escenarios existentes.

14.4.3 Pruebas de la primera etapa

Como parte de la implementación de la primera etapa, hemos desarrollado dos clases de prueba: `LocationTest` y `TaxiTest`. La primera comprueba la funcionalidad básica de la clase `Location`, que resulta fundamental para el correcto funcionamiento de los vehículos. La segunda está diseñada para comprobar que se recoge a un pasajero y se le lleva hasta su destino en el número correcto de pasos, y que el taxi pasa a estar libre inmediatamente después. Para desarrollar el segundo conjunto de pruebas, se amplió la clase `Location` con el método `distance`, que proporciona el número de pasos requeridos para moverse entre dos lugares.[1]

[1] Anticipamos que esto tendrá un uso más amplio posteriormente en el desarrollo de la aplicación, ya que debería permitir a la compañía planificar los vehículos basándose en cual está más cerca del punto de recogida.

Durante la operación normal, la aplicación se ejecuta en silencio, y al no disponer de una GUI no existe una manera visual de monitorizar el progreso de un taxi. Una solución sería añadir instrucciones de impresión a los métodos fundamentales de clases tales como `Taxi` y `TaxiCompany`. Sin embargo, BlueJ ofrece la alternativa de establecer un punto de interrupción dentro del método `act`, de por ejemplo la clase `Taxi`. Esto permitiría "observar" el movimiento de un taxi por inspección.

Habiendo alcanzado un nivel de confianza razonable en el estado actual de la implementación, hemos dejado simplemente instrucciones de impresión en los métodos de notificación de `TaxiCompany` para proporcionar un mínimo de realimentación al usuario.

Como testimonio de la ventaja que tiene desarrollar las pruebas junto con la implementación, merece la pena resaltar que las clases de prueba existentes nos permitieron identificar y corregir dos errores graves de nuestro código.

Ejercicio 14.24 Revise las pruebas implementadas en las clases de prueba de *taxi-company-stage-one*. ¿Cree que sería posible utilizarlas como pruebas de regresión en las siguientes etapas, o hará falta hacer en ellas modificaciones sustanciales?

Ejercicio 14.25 Implemente las pruebas y clases de prueba adicionales que crea que son necesarias para incrementar nuestro grado de confianza en la implementación actual. Corrija cualquier error que descubra durante el proceso.

14.4.4 Una etapa posterior del desarrollo

No es nuestra intención analizar en detalle el proceso completo del desarrollo de la aplicación de la compañía de taxis, ya que el lector no sacaría mucho de ese análisis. En lugar de ello, lo que haremos será presentar brevemente la aplicación en una etapa posterior y animarle a que complete el resto por sí mismo a partir de ahí.

Esta etapa más avanzada puede encontrarse en el proyecto *taxi-company-later-stage*. Gestiona múltiples taxis y pasajeros y una GUI proporciona una vista progresiva de los movimientos de ambos (Figura 14.2). He aquí un esbozo de algunos de los desarrollos más importantes de esta versión, comparada con la anterior.

- Una clase `Simulation` se encarga ahora de gestionar los actores, de forma bastante similar a como se hacía en el proyecto *foxes-and-rabbits*. Los actores son los vehículos, el origen de pasajeros y una GUI proporcionada por la clase `CityGUI`. Después de cada paso, la simulación hace una pausa durante un corto periodo de tiempo para que la GUI no cambie con demasiada rapidez.
- La necesidad de algo parecido a la clase `City` fue identificada durante el desarrollo de la primera etapa. El objeto `City` define las dimensiones de la cuadrícula de la ciudad y almacena una colección de todos los elementos de interés contenidos en la ciudad (los vehículos y los pasajeros).
- Los elementos contenidos en la ciudad pueden implementar opcionalmente la interfaz `DrawableItem`, que permite a la GUI visualizarlos. En la subcarpeta `images` de la carpeta del proyecto se proporcionan, con este objetivo, imágenes de los vehículos y de las personas.

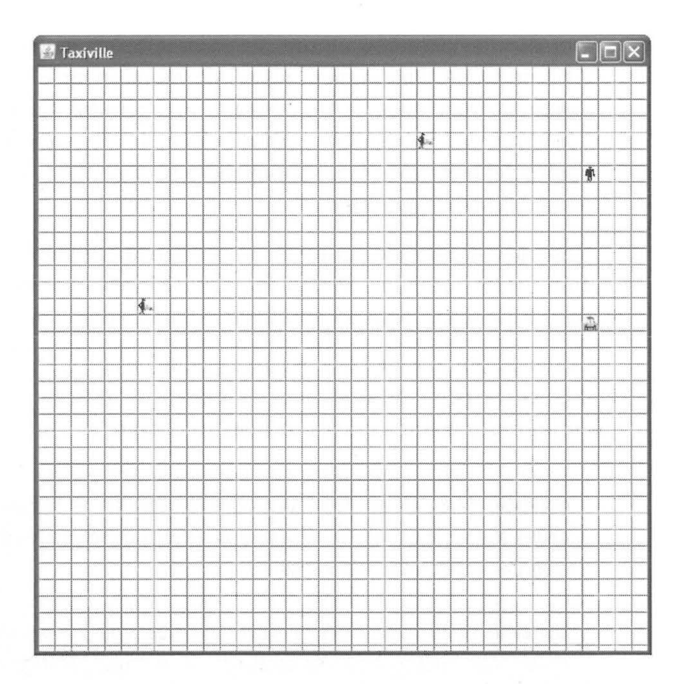


Figura 14.2
Una visualización de la ciudad.

- La clase `Taxi` implementa la interfaz `DrawableItem`. Devuelve imágenes alternativas a la GUI, dependiendo de si está ocupado o vacío. Hay disponibles archivos de imagen en la carpeta `images` para poder hacer lo mismo con una lanzadera.
- La clase `PassengerSource` se ha refactorizado de manera significativa con respecto a la versión anterior, para que encaje mejor en su papel de actor. Además, mantiene un recuento de las recogidas de pasajeros que no han podido ser satisfechas para poder efectuar un análisis estadístico.
- La clase `TaxiCompany` es responsable de crear los taxis que se van a utilizar en la simulación.

A medida que explore el código fuente del proyecto *taxi-company-later-stage*, encontrará ilustraciones de muchos de los temas que hemos cubierto en la segunda mitad de este libro: herencia, polimorfismo, clases abstractas, interfaces y tratamiento de errores.

Ejercicio 14.26 Añada a cada clase comprobaciones de coherencia basadas en aserciones y en generación de excepciones, para proteger las clases frente a un uso inapropiado. Por ejemplo, asegúrese de no crear nunca un `Passenger` con lugares de recogida y de destino coincidentes; asegúrese también de que nunca se encargue a un taxi ir a un lugar de recogida cuando ya se está dirigiendo a un lugar de destino, etc.

Ejercicio 14.27 Haga las modificaciones necesarias para informar sobre los datos estadísticos que están siendo recopilados por los taxis y el origen de pasajeros; debe informarse también del tiempo de inactividad de los taxis y de las recogidas que no se han podido realizar. Experimente con diferentes cantidades de taxis para ver como varía la relación entre estos dos conjuntos de datos.

Ejercicio 14.28 Adapte las clases de vehículos para que se mantengan registros del tiempo invertido en desplazarse a los lugares de recogida y a los destinos de los pasajeros. ¿Detecta ahí algún posible conflicto para las lanzaderas?

14.4.5 Ideas adicionales de desarrollo

La versión de la aplicación proporcionada en el proyecto *taxi-company-later-stage* representa un punto significativo del desarrollo, dentro del proceso de creación de la implementación completa. Sin embargo, todavía son muchas las cosas que se pueden añadir. Por ejemplo, apenas hemos desarrollado la clase `Shuttle`, por lo que se encontrará con una multitud de desafíos a la hora de completar su implementación. La principal diferencia entre las lanzaderas y los taxis es que una lanzadera tiene que preocuparse de múltiples pasajeros, mientras que un taxi solo se ocupa de un pasajero cada vez. El hecho de que una lanzadera ya esté transportando un pasajero no debe impedir que se la envíe a recoger a otro. De forma similar, si ya está dirigiéndose a recoger a un pasajero, puede seguir aceptando solicitudes de recogida adicionales. Estos problemas plantean cuestiones acerca de cómo organiza una lanzadera sus prioridades. ¿Podría resultar que un pasajero termine siendo llevado de un sitio a otro mientras la lanzadera responde a otras solicitudes, sin que el pasajero llegue nunca a su destino? ¿Qué quiere decir que una lanzadera no esté libre? ¿Quiere decir que está llena de pasajeros o que ha recibido suficientes solicitudes de recogida como para llenarla? Suponga que al menos uno de los pasajeros que ya han sido recogidos llega a su destino antes de llegar al punto de la última recogida: ¿quiere eso decir que la lanzadera podría aceptar más solicitudes de recogida que su capacidad física?

Otra área de desarrollo posterior es la referida a la planificación de vehículos. La compañía de taxis no opera en la actualidad los vehículos de una forma especialmente inteligente. ¿Cómo debería decidir qué vehículo enviar para recoger un pasajero, cuando hay más de un vehículo disponible? No se hace ningún intento de asignar los vehículos basándose en la distancia con respecto al lugar de recogida. La compañía podría utilizar el método `distance` de la clase `Location` para determinar cuál es el vehículo libre más próximo a un lugar de recogida. ¿Tendría esto una influencia significativa sobre el tiempo medio de espera de los pasajeros? ¿Cómo podrían recopilarse datos acerca de cuánto tiempo pasan los pasajeros esperando a ser recogidos? ¿Y qué le parece la idea de hacer que los taxis vacíos se desplacen hasta una ubicación central, listos para la siguiente recogida, con el fin de reducir los tiempos de espera potenciales? ¿Influye el tamaño de la ciudad sobre la efectividad de esta solución? Por ejemplo, en una ciudad grande, ¿es mejor hacer que los taxis vacíos estén espaciados uno con respecto a otro, en lugar de que todos se concentren en el centro?

¿Podría utilizarse la simulación para modelar a una serie de compañías de taxis competidoras que operen en la misma área de la ciudad? Podrían crearse múltiples objetos `TaxiCompany` y el origen de pasajeros podría asignarles los pasajeros de forma competitiva, basándose en lo rápido que cada uno pudiera recoger a cada pasajero. ¿Cree que este cambio es demasiado grande como para que pueda llevarse a cabo con la aplicación existente?

14.4.6 Reutilización

Actualmente, nuestro objetivo era simular la operación de una serie de vehículos, con el fin de evaluar la viabilidad comercial del proyecto de ampliación de un negocio a una nueva área de la ciudad. Puede que haya observado que hay partes importantes de la aplicación que pueden seguir siendo útiles una vez puesta en marcha la ampliación.

Suponiendo que desarrollemos un algoritmo de planificación inteligente para que nuestra simulación decida qué vehículo debe aceptar cada petición, o asumiendo que hayamos desarrollado un buen esquema para decidir a dónde enviar los vehículos con el fin de que

esperen mientras están inactivos, podríamos decidir emplear los mismos algoritmos cuando la compañía empiece de verdad a operar en la nueva área. La representación visual de la posición de cada vehículo también podría servir de ayuda.

En otras palabras, existe un gran potencial para transformar la simulación de la compañía de taxis en un sistema de gestión de taxis, utilizado para ayudar a la compañía real en sus operaciones. La estructura de la aplicación cambiaría, por supuesto: el programa no controlaría y movería los taxis, sino que en su lugar anotaría sus posiciones, que podría recibir gracias a los receptores GPS (*Global Positioning System*, Sistema de posicionamiento global) equipados en cada vehículo. Sin embargo, muchas de las clases desarrolladas para la simulación podrían reutilizarse con muy pocos cambios o ninguno. Esto ilustra la potencia de reutilización que obtenemos al disponer de una buena estructura de clases y un buen diseño de clases.

14.5 Otro ejemplo

Hay muchos otros proyectos que podríamos realizar siguiendo líneas similares a las de la aplicación de la compañía de taxis. Un alternativa muy popular es la de cómo planificar los ascensores en un gran edificio. Aquí, la coordinación entre los ascensores pasa a ser particularmente importante. Además, dentro de un edificio cerrado, puede ser posible estimar el número de personas que hay en cada planta y prever así la demanda. También hay que tener en cuenta comportamientos relacionados con la hora del día: llegada de personas por la mañana, salida de personas por la tarde y picos locales de actividad en torno a las horas de la comida.

Utilice la técnica que hemos explicado en este capítulo para implementar una simulación de un edificio con uno o más ascensores.

14.6 A partir de aquí

No podemos ir mucho más allá a la hora de presentarle nuestras propias ideas de proyectos y demostrarle cómo las desarrollaríamos nosotros. Si desarrolla sus propias ideas de proyectos y las implementa a su manera, comprobará que puede llegar mucho mas lejos. Elija un tema de su interés, y trabaje las distintas etapas que hemos esbozado: analice el problema, desarrolle algunos escenarios, esboce un diseño, planifique algunas etapas de implementación y luego póngase manos a la obra.

Diseñar e implementar programas es una actividad excitante y creativa. Como cualquier otra actividad que merece la pena, requiere tiempo y práctica llegar a dominarla. Por tanto, no se deje desanimar si sus primeros esfuerzos parecen requerir un tiempo desmedido o están llenos de errores. Esto es normal, y ya irá mejorando gradualmente con la experiencia. No sea demasiado ambicioso para empezar, y tenga por seguro que necesitará revisar sus ideas a medida que vaya progresando; eso forma parte del proceso natural de aprendizaje.

Pero por encima de todo: ¡Diviértase!

APÉNDICE

A Cómo trabajar con un proyecto BlueJ

A.1 Instalación de BlueJ

Para trabajar con BlueJ, es necesario instalar un kit de desarrollo Java (JDK, *Java Development Kit*) y el entorno de BlueJ.

Puede encontrar el JDK e instrucciones de instalación detalladas en el CD de acompañamiento de este libro o en `http://www.oracle.com/technetwork/java/javase/overview/index.html`

El entorno BlueJ así como las instrucciones de instalación están disponibles en el CD de acompañamiento del libro o en `http://www.bluej.org`

A.2 Cómo abrir un proyecto

Para poder utilizar cualquiera de los proyectos de ejemplo incluidos en el CD de acompañamiento del libro, es necesario copiar dichos proyectos en un disco habilitado para escritura. Los proyectos BlueJ pueden abrirse pero no ejecutarse desde un CD (para ejecutarlos, BlueJ necesita escribir en la carpeta del proyecto). Por tanto, normalmente no será posible utilizar los proyectos directamente desde el CD.

Por tanto, lo más sencillo es copiar la carpeta que contiene todos los proyectos del libro (denominada *projects*) en el disco duro. Después de haber instalado e iniciado BlueJ haciendo doble clic en su icono, seleccione *Open...* (Abrir...) en el menú *Project* (Proyecto). Sitúese en la carpeta *projects* y seleccione un proyecto. (Puede tener abiertos varios proyectos simultáneamente.) Cada una de las carpetas de proyecto contiene un archivo `bluej.project` sobre el que, y se está asociado con BlueJ, puede hacerse doble clic para abrir directamente el proyecto.

Para obtener más información sobre el uso de BlueJ consulte el tutorial de BlueJ. Este tutorial está disponible en el CD del libro y se puede acceder a él también a través del elemento *BlueJ Tutorial* disponible en el menú *Help* (Ayuda) de BlueJ.

A.3 El depurador de BlueJ

Puede encontrar información sobre la utilización del depurador de BlueJ en el Apéndice F y en el tutorial de BlueJ. El tutorial está disponible en el CD de acompañamiento del

libro y también se puede acceder a él a través del elemento *BlueJ Tutorial* del menú *Help* de BlueJ.

A.4 Contenido del CD

En el CD de acompañamiento del libro encontrará los siguientes archivos y directorios:

Carpeta	Comentario
bluej/	*El sistema BlueJ y la documentación correspondiente.*
bluejsetup-305.exe	*El instalador de BlueJ para Microsoft Windows (todas las versiones).*
BlueJ-305.zip	*BlueJ para Mac OS X.*
bluej-305.deb	*BlueJ para sistema basados en Debian (Debian, Ubuntu).*
bluej-305.jar	*BlueJ para otros sistemas.*
tutorial.pdf	*El tutorial de BlueJ.*
testing-tutorial.pdf	*Un tutorial que presenta las herramientas de prueba.*
teamwork-tutorial.pdf	*Tutorial para el uso de las herramientas de trabajo en equipo.*
repository-setup.pdf	*Información sobre cómo configurar un CVS o un repositorio de subversiones para soporte de trabajo en equipo.*
ReadMe.htm	*Documentación del CD. Abra este archivo en un explorador web para leerlo. Contiene una panorámica del contenido del CD, instrucciones de instalación y otras indicaciones útiles.*
jdk/	*Contiene sistemas Java (JDK) para varios sistemas operativos.*
linux/	*Instalador de JDK para Linux.*
solaris/	*Instalador de JDK para Solaris.*
windows/	*Instalador de JDK para Microsoft Windows (todas las versiones).*
jdk-doc/	*Contiene la documentación de la librería Java. Es un único archivo zip. Copie este archivo en su disco duro y descomprímalo para poder utilizar la documentación.*
projects/	*Contiene todos los proyectos explicados en el libro. Copie la carpeta completa en su disco duro antes de utilizar los proyectos. Contiene subcarpetas para cada capítulo.*

A.5 Configuración de BlueJ

Muchas de las opciones de configuración de BlueJ pueden ajustarse para satisfacer cualquier necesidad personal que tengamos. Algunas opciones de configuración están disponibles a

través del cuadro de diálogo *Preferences* (Preferencias) en el sistema BlueJ, pero se puede acceder a muchas otras opciones de configuración editando el "archivo de definiciones BlueJ". Este archivo se encuentra en *<bluej_home>/lib/bluej.defs*, donde *<bluej_home>* es la carpeta donde esté instalado el sistema BlueJ.[1]

Los detalles de configuración se explican en el archivo *Tips archive* en el sitio web de BlueJ. Puede acceder a él a través de la dirección:

```
http://www.bluej.org/help/archive.html
```

A continuación se indican algunas de las cosas más comunes que se suelen configurar. Puede encontrar muchas más opciones de configuración leyendo el archivo *bluej.defs*.

A.6 Cómo cambiar el idioma de la interfaz

Puede cambiar el idioma de la interfaz para utilizar uno de los lenguajes disponibles. Para ello, abra el archivo `bluej.defs` y localice la línea siguiente:

```
bluej.language=english
```

Cámbiela a alguno de los otros idiomas disponibles. Por ejemplo:

```
bluej.language=spanish
```

Los comentarios del archivo enumeran todos los idiomas disponibles, entre lo que se incluyen afrikaans, catalán, chino, checo, danés, holandés, inglés, francés, alemán, griego, italiano, japonés, coreano, portugués, ruso, eslovaco, español y sueco.

A.7 Utilización de la documentación local de la API

Puede utilizar una copia local de la documentación de la librería de clases Java (API). De esta forma, el acceso a la documentación será más rápido y podrá utilizarla sin necesidad de estar conectado a la red. Para hacer esto, copie el archivo de documentación de Java desde el CD del libro (un archivo zip) y descomprímalo en la ubicación donde desee almacenar la documentación Java. Esto creará una carpeta denominada *jdk-7-api-doc*.

A continuación, abra un explorador web y utilizando la función *Abrir archivo* (u otra equivalente), abra el archivo *index.html* contenido en esta carpeta. Una vez que la vista de la API se muestre correctamente en el explorador, copie la URL (dirección web) del campo de dirección de su explorador, abra BlueJ, abra el cuadro de diálogo *Preferences*, acceda a la pestaña *Miscellaneous* y pegue la URL que ha copiado en el campo etiquetado como *JDK documentation URL*. A partir de ahí, al utilizar el elemento *Java Class Libraries* del menú *Help* debería abrirse la copia local.

A.8 Modificación de las plantillas para nuevas clases

Cuando cree una nueva clase,el código de la nueva clase se configura con un texto predeterminado. Este texto está extraído de una plantilla y puede cambiarse para adaptarlo a nuestras preferencias. Las plantillas están almacenadas en las carpetas:

---

[1] En Mac OS, el archivo *bluej.defs* se encuentra dentro del paquete de la aplicación. Consulte el archivo de consejos (*Tips archive*) para ver instrucciones acerca de cómo localizarlo.

<bluej_home>/lib/<language>/templates/

y

<bluej_home>/lib/<language>/templates/newclass/

donde *<bluej_home>* es la carpeta de instalación de BlueJ y *<language>* es la opción de idioma que actualmente esté empleando (por ejemplo, *english*).

Los archivos de plantilla son archivos de texto puro y pueden editarse con cualquier editor de texto estándar.

APÉNDICE

B Tipos de datos Java

El sistema de tipos de Java está basado en dos clases distintas de tipos: tipos primitivos y tipos de objeto.

Los tipos primitivos se almacenan directamente en variables y están dotados de una semántica de valor (los valores se copian al ser asignados a otra variable). Los tipos primitivos no están asociados con clases y no tienen métodos.

Por contraste, un tipo de objeto se manipula almacenando una referencia al objeto (no al propio objeto). Cuando se asigna a una variable, solo se copia la referencia, no el objeto.

B.1 Tipos primitivos

La siguiente tabla enumera todos los tipos primitivos del lenguaje Java:

Nombre del tipo	Descripción	Literales de ejemplo		
Números enteros				
`byte`	entero con tamaño de un byte (8 bits)	`24`	`-2`	
`short`	entero corto (16 bits)	`137`	`-119`	
`int`	entero (32 bits)	`5409`	`-2003`	
`long`	entero largo (64 bits)	`423266353L`	`55L`	
Números reales				
`float`	coma flotante de simple precisión	`43.889F`		
`double`	coma flotante de doble precisión	`45.63`	`2.4e5`	
Otros tipos				
`char`	un único carácter (16 bit)	`'m'`	`'?'`	`'\u00F6'`
`boolean`	un valor booleano (*true* o *false*)	`true`	`false`	

Notas:

- Un número sin punto decimal se interpreta generalmente como un `int`, pero es convertido automáticamente a tipos `byte`, `short` o `long` cuando se lo asigna (si el valor es apropiado). Puede declararse un literal como `long` incluyendo una `L` después del número. (También funciona con `l`, `L` minúscula, pero se recomienda evitar su uso porque puede confundirse fácilmente con un uno (`1`).)

- Un número con punto decimal es de tipo `double`. Puede especificarse un literal `float` poniendo una `F` o una `f` después del número.
- Un carácter puede escribirse como un único carácter Unicode entre comillas simples o como un valor Unicode de cuatro dígitos precedido por "`\u`".
- Los dos literales booleanos son `true` y `false`.

Puesto que las variables de los tipos primitivos no hacen referencia a los objetos, no hay métodos asociados con los tipos primitivos. Sin embargo, cuando se utilizan en un contexto donde haga falta un tipo de objeto, puede utilizarse la característica de *autoboxing* para convertir un valor primitivo en su objeto correspondiente. Consulte la Sección B.4 para conocer más detalles.

La siguiente tabla detalla los valores mínimo y máximo disponibles para los tipos numé ricos.

Nombre del tipo	Mínimo	Máximo
byte	-128	127
short	-32768	32767
int	-2147483648	2147483647
long	-9223372036854775808	9223372036854775807
	Mínimo positivo	**Máximo positivo**
float	1.4e-45	3.4028235e38
double	4.9e-324	1.7976931348623157e308

B.2 Conversión de tipos primitivos

En ocasiones, es necesario convertir un valor de un tipo primitivo a un valor de otro tipo primitivo; normalmente, un valor de un tipo con un cierto rango de valores a otro con un rango más pequeño. Este proceso se denomina *cast*. La conversión de tipos implica casi siempre pérdida de información; por ejemplo, al convertir de un tipo de coma flotante a un tipo entero. En Java se permite la conversión entre tipos numéricos, pero no es posible convertir un valor de tipo `boolean` en ningún otro tipo con un *cast*, o viceversa.

El operador de cast consta del nombre de un tipo primitivo escrito entre paréntesis delante de una variable o expresión. Por ejemplo,

```
int val = (int) mean;
```

Si `mean` es una variable de tipo `double` que contiene el valor 3.9, entonces la instrucción anterior almacenaría el valor entero 3 (conversión por truncamiento) en la variable `val`.

B.3 Tipos de objeto

Todos los tipos no enumerados en la Sección B.1, *Tipos primitivos*, son tipos de objeto. Entre estos se incluyen las clases y los tipos de interfaz de la librería Java (como `String`), así como los tipos definidos por el usuario.

Una variable de un tipo de objeto almacena una referencia (o "puntero") a un objeto. Las asignaciones y el paso de parámetros tienen semántica de referencia (es decir, se copia la referencia, no el objeto). Después de asignar una variable a otra, ambas variables harán referencia al mismo objeto. Decimos en este caso que ambas variables son alias del mismo objeto. Esta regla se aplica en la asignación simple entre variables, pero también al pasar a un método un objeto como parámetro real. En consecuencia, los cambios de estado efectuados en un objeto a través de un parámetro formal serán persistentes y se conservarán en el parámetro real después de haberse completado el método.

Las clases son las plantillas de los objetos, encargándose de definir los campos y métodos que cada instancia posee.

Las matrices se comportan como tipos de objeto; también tienen semántica de referencia. No hay definición de clase para las matrices.

B.4 Clases envoltorio

Todo tipo primitivo en Java tiene una clase envoltorio correspondiente que representa el mismo tipo, pero que es un tipo de objeto real. Esto hace posible utilizar valores de los tipos primitivos en ocasiones en las que se exigen tipos de objeto, a través de un proceso conocido con el nombre de *autoboxing*. La siguiente tabla enumera los tipos primitivos y su correspondiente tipo envoltorio del paquete `java.lang`. A excepción de `Integer` y `Character`, los nombres de las clases envoltorio coinciden con los nombres de los tipos primitivos, pero escribiéndose la primera letra del nombre en mayúscula.

Tipo primitivo	Tipo envoltorio
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Cuando se utiliza un valor de un tipo primitivo en un contexto que requiere un tipo de objeto, el compilador utiliza el mecanismo de *autoboxing* para envolver automáticamente el valor de tipo primitivo en un objeto envoltorio apropiado. Esto significa, por ejemplo, que pueden añadirse directamente valores de tipo primitivo a una colección. La operación inversa –*unboxing*– también se realiza automáticamente cuando se emplea un objeto de tipo envoltorio en un contexto que requiere un valor del correspondiente tipo primitivo.

B.5 *Cast* de tipos de objeto

Puesto que un objeto puede pertenecer a una jerarquía de herencia de tipos, en ocasiones es necesario convertir una referencia de objeto en una referencia a un subtipo situado más abajo

en la jerarquía de herencia. Este proceso se denomina *casting* (o *downcasting*). El operador de cast está compuesto por el nombre de una clase o tipo de interfaz escrito entre paréntesis delante de una variable o una expresión. Por ejemplo,

```
Car c = (Car) veh;
```

Si el tipo declarado (es decir, estático) de la variable `veh` es `Vehicle` y `Car` es una subclase de `Vehicle`, entonces esta instrucción podrá compilarse correctamente. En tiempo de ejecución se hace otra comprobación independiente, para garantizar que el objeto al que hace referencia `veh` sea realmente un `Car` y no una instancia de un subtipo distinto.

Es importante comprender que el *cast* entre tipo de objetos es completamente distinto de la conversión entre tipos primitivos (véase la Sección B.2). En particular, el *cast* entre tipos de objetos no implica *ninguna modificación* del objeto implicado. Se trata simplemente de una forma de obtener acceso a una información de tipo que ya es cierta para ese objeto; es decir, que forma parte de su tipo dinámico completo.

APÉNDICE

C Operadores

C.1 Expresiones aritméticas

Java tiene un considerable número de operadores disponibles tanto para expresiones aritméticas como lógicas. La Tabla C.1 muestra todo lo que está clasificado como operador, incluyendo el *cast* de tipos y el paso de parámetros. La mayor parte de los operadores son operadores binarios (con un operando izquierdo y otro derecho) u operadores unarios (con un solo operando). Las principales operaciones aritméticas binarias son:

- `+` *suma*
- `–` *resta*
- `*` *multiplicación*
- `/` *división*
- `%` *módulo* o *resto después de la división*

El resultado tanto de la operación de división como de la de módulo depende de si sus operandos son valores enteros o de coma flotante. Entre dos valores enteros, la división da un resultado entero, descartándose cualquier resto, pero entre valores de coma flotante, el resultado es un valor de coma flotante:

`5 / 3` da un resultado de `1`
`5.0 / 3` da un resultado de `1.6666666666666667`

(Observe que basta con que uno de los operadores tenga un tipo de coma flotante para que se genere un resultado de coma flotante.)

Cuando aparece más de un operador en una expresión, hay que aplicar las *reglas de precedencia* para dilucidar el orden de aplicación de los operadores. En la Tabla C.1, los operadores con mayor precedencia aparecen en la parte superior, así podemos ver, por ejemplo, que la multiplicación, la división y el módulo tienen precedencia sobre la suma y la resta. Esto significa que los dos siguientes ejemplos darían como resultado `100`:

```
51 * 3 - 53
154 – 2 * 27
```

Los operadores binarios con el mismo nivel de precedencia se evalúan de izquierda a derecha, mientras que los operadores unarios con el mismo nivel de precedencia se evalúan de derecha a izquierda.

Cuando es necesario alterar el orden normal de evaluación, pueden utilizarse paréntesis. De este modo, los dos ejemplos siguientes nos darían como resultado `100`:

```
(205 - 5) / 2
2 * (47 + 3)
```

Los principales operadores unarios son: –, `!`, ++, – – , `[]` y `new`. Observe que ++ y – – aparecen en cada una de las dos primeras filas de la Tabla C.1. Los que están en la primera fila admiten un único operando a su izquierda, mientras que los que están en la segunda fila admiten un único operando a su derecha.

Tabla C.1 Operadores Java, mostrándose en la parte superior los de mayor precedencia.

[]	.	++	--	(*parámetros*)							
++	--	+	-	!	~						
new	(*cast*)										
*	/	%									
+	-										
<<	>>	>>>									
<	>	>=	<=	instanceof							
==	!=										
&											
^											
|											
&&											
||											
?:											
=	+=	-=	*=	/=	%=	>>=	<<=	>>>=	&=	|=	^=

C.2 Expresiones booleanas

En las expresiones booleanas se utilizan operadores para combinar operandos, con el fin de generar un valor que puede ser `true` o `false`. Dichas expresiones suelen encontrarse en las expresiones de comprobación de las instrucciones if y de los bucles.

Normalmente, los operadores relacionales combinan una pareja de operandos aritméticos, aunque las pruebas de igualdad y desigualdad pueden también aplicarse a referencias de objetos. Los operadores relacionales Java son:

==	igual que	!=	distinto de
<	menor que	<=	menor o igual que
>	mayor que	>=	mayor o igual que

Los operadores lógicos binarios combinan dos expresiones booleanas para generar otro valor booleano. Los operadores son:

`&&` and
`||` or
`^` or exclusiva

Además,

`!` not

admite una única expresión booleana y la cambia de `true` a `false`, y viceversa.

C.3 Operadores de cortocircuito

Tanto `&&` como `||` son ligeramente inusuales, en lo que respecta a la forma en que se aplican. Si el operando izquierdo de `&&` es `false`, entonces el valor del operando derecho es irrelevante y no será evaluado. Del mismo modo, si el operando izquierdo de `||` es `true`, entonces no se evalúa el operando derecho. Debido a ello, se les conoce con el nombre de operadores de cortocircuito.



D.1 Estructuras de control

Las estructuras de control afectan al orden en que se ejecutan las instrucciones. Existen dos categorías principales: *instrucciones de selección* y *bucles*.

Una instrucción de selección proporciona un punto de decisión en el que se realiza una elección para seguir una ruta a través del cuerpo de un método o constructor en lugar de otra ruta. Una *instrucción if-else* implica una decisión entre dos conjuntos diferentes de instrucciones, mientras que una *instrucción switch* permite seleccionar una única opción entre varias.

Los bucles ofrecen la opción de repetir instrucciones, un número definido o indefinido de veces. La repetición de un número definido de veces es ejemplificada por el *bucle for-each* y por el *bucle for*, mientras que la repetición indefinida está tipificada por el *bucle while* y el *bucle do*.

En la práctica, hay que tener presente que las excepciones a esta caracterización son bastante comunes. Por ejemplo, puede utilizarse una *instrucción if-else* para seleccionar entre varios conjuntos alternativos de instrucciones siempre que la parte *else* contenga una instrucción if-else anidada, mientras que un bucle for puede emplearse para repetir un conjunto de instrucciones un número indefinido de veces.

D.2 Instrucciones de selección

D.2.1 if-else

La *instrucción if-else* tiene dos formas principales, estando ambas controladas por la evaluación de una expresión booleana:

```
if(expresión) {
    instrucciones
}
```

```
if(expresión) {
    instrucciones
}
else {
    instrucciones
}
```

En la primera forma, se utiliza la expresión booleana para decidir si ejecutar o no las instrucciones. En la segunda forma, se emplea la expresión para elegir entre dos conjuntos de instrucciones alternativos, ejecutándose solo uno de ellos.

Ejemplos:

```
if(field.size() == 0) {
    System.out.println("The field is empty.");
}
```

```
if(number < 0) {
    reportError();
}
else {
   processNumber(number);
}
```

Es muy común encadenar instrucciones if-else colocando un segundo *if-else* en la parte *else* del primero. Esto puede continuarse cualquier número de veces que se desee. Es conveniente incluir siempre una parte *else* final.

```
if(n < 0) {
    handleNegative();
}
else if(number == 0) {
    handleZero();
}
else {
   handlePositive();
}
```

D.2.2 switch

La *instrucción switch* utiliza un único valor para seleccionar uno de entre un número arbitrario de casos. Dos posibles patrones de uso son los siguientes:

```
switch(expresión) {
    case value: instrucciones;
                break;
    case value: instrucciones;
                break;
    es posible tener más casos
    default: instrucciones;
             break;
}
```

```
switch(expresión) {
      case valor1:
      case valor2:
      case valor3:
         instrucciones;
              break;
      case valor4:
      case valor5:
         instrucciones;
         break;
      es posible tener más casos
      default:
         instrucciones;
         break;
}
```

Notas:

- Una instrucción switch puede tener cualquier número de etiquetas `case`.
- La instrucción `break` después de cada caso es necesaria; si no se incluye, la ejecución continúa con las instrucciones correspondientes a la siguiente etiqueta. La segunda forma mostrada anteriormente hace uso de esta característica. En este caso, los tres primeros valores ejecutarán la primera sección de *instrucciones*, mientras que los valores cuarto y quinto ejecutarán la segunda de sección de *instrucciones*.
- El caso `default` es opcional. Si no se proporciona este caso, que se corresponde con la opción predeterminada, puede ser que no se ejecute ningún caso.
- La instrucción `break` después del caso predeterminado (o del último caso, si es que no hay ninguno predeterminado) no es necesaria, aunque se considera que el incluirla refleja un buen estilo de programación.
- A partir de Java 7, la expresión utilizada como selector y en las etiquetas de los casos puede ser una expresión de cadena.

Ejemplos:

```
switch(day) {
    case 1:  dayString = "Monday";
             break;
    case 2:  dayString = "Tuesday";
             break;
    case 3:  dayString = "Wednesday";
             break;
    case 4:  dayString = "Thursday";
             break;
    case 5:  dayString = "Friday";
             break;
    case 6:  dayString = "Saturday";
             break;
    case 7:  dayString = "Sunday";
             break;
    default: dayString = "invalid day";
             break;
}

switch(dow.toLowerCase()) {
    case "mon":
    case "tue":
    case "wed":
    case "thu":
    case "fri":
        goToWork();
        break;
    case "sat":
    case "sun":
        stayInBed();
        break;
}
```

D.3 Bucles

Java proporciona tres bucles: *while*, *do-while* y *for*. El *bucle for* tiene dos formas. Tanto *while* como *do-while* están bien adaptados a la realización de iteraciones indefinidas. El bucle for-each está pensado para realizar una iteración definida a través de una colección, mientras que el bucle for cae en cierto modo en medio de ambos extremos. Excepto en el caso del bucle for-each, la repetición en cada bucle está controlada por una expresión booleana.

D.3.1 *while*

El *bucle while* ejecuta un bloque de instrucciones mientras una cierta expresión se evalúa como *true*. La expresión se evalúa *antes* de la ejecución del cuerpo del bucle, por lo que es posible que ese cuerpo se ejecute cero veces (es decir, que no se llegue a ejecutar). Esta capacidad es una importante característica del bucle while.

```
while(expresión) {
        instrucciones
}
```

Ejemplos:

```
System.out.print("Please enter a filename: ");
input = readInput();
while(input == null) {
    System.out.print("Please try again: ");
    input = readInput();
}

int index = 0;
boolean found = false;
while(!found && index < list.size()) {
    if(list.get(index).equals(item)) {
       found = true;
    }
    else {
       index++;
    }
}
```

D.3.2 *do-while*

El *bucle do-while* ejecuta un bloque de instrucciones mientras que una cierta expresión se evalúe como *true*. La expresión se comprueba *después* de la ejecución del cuerpo del bucle, por lo que el cuerpo siempre se ejecuta al menos una vez. Esta es una diferencia importante con respecto al bucle while.

```
do {
    instrucciones
} while(expresión);
```

Ejemplo:

```
do {
   System.out.print("Please enter a filename: ");
   input = readInput();
} while(input == null);
```

D.3.3 *for*

El *bucle for* tiene dos formas diferentes. La primera se conoce también con el nombre de *bucle for-each* y se utiliza exclusivamente para iterar a través de los elementos de una colección. A la variable del bucle se le asigna el valor de los elementos sucesivos de la colección en cada iteración del bucle.

```
for(declaración-variable: colección) {
    instrucciones
}
```

Ejemplo:

```
for(String note : list) {
    System.out.println(note);
}
```

No hay disponible ningún valor de índice asociado a los elementos de la colección. Un bucle for-each no se puede utilizar si hay que modificar la colección mientras se itera a través de ella.

La segunda forma de bucle for se ejecuta mientras que una *condición* se evalúe como *true*. Antes de que se inicie el bucle, se ejecuta una *instrucción de inicialización* exactamente una vez. La *condición* se evalúa antes de cada ejecución del cuerpo del bucle, por lo que las instrucciones del cuerpo del bucle pueden ejecutarse cero veces. Después de cada ejecución del cuerpo del bucle se ejecuta una *instrucción de incremento*.

```
for(inicialización; condición; incremento) {
    instrucciones
}
```

Ejemplo:

```
for(int i = 0; i < text.size(); i++) {
    System.out.println(text.get(i));
}
```

Ambos tipos de bucle for se utilizan comúnmente para ejecutar el cuerpo del bucle un número definido de veces; por ejemplo, una vez por cada elemento de una colección, aunque un bucle for está en la práctica más próximo a un bucle while que a un bucle for-each.

D.4 Excepciones

La generación y captura de excepciones proporciona otra pareja de estructuras que permiten alterar el control de flujo. Sin embargo, las excepciones se emplean principalmente para prever y tratar situaciones de error, más que para el flujo normal de control.

```
try {
    instrucciones
}
catch(nombre tipo-excepción) {
      instrucciones
}
finally {
    instrucciones
}
```

Ejemplo:

```
try {
    FileWriter writer = new FileWriter("foo.txt");
    writer.write(text);
    writer.close();
}
catch(IOException e) {
    Debug.reportError("Writing text to file failed.");
    Debug.reportError("The exception is: " + e);
}
```

Una instrucción de excepción puede tener cualquier número de cláusulas *catch*. Estas cláusulas se evalúan por orden de aparición y solo se ejecutará la primera cláusula para la que se encuentre una correspondencia. (La cláusula tendrá una correspondencia si el tipo dinámico del objeto excepción que se ha generado es compatible, desde el punto de vista de la asignación, con el tipo declarado de excepción en la cláusula catch.) La cláusula *finally* es opcional.

Java 7 ha introducido dos adiciones principales a la instrucción try: la multicaptura y el try con recursos, también denominado gestión automática de recursos (ARM, *Automatic Resource Management*).

Se pueden tratar varias excepciones en la misma cláusula catch escribiendo la lista de tipos de excepción, separados por el símbolo "|".

Ejemplo:

```
try {
    ...
    var.doSomething();
    ...
}
catch(EOFException | FileNotFoundException e) {
    ...
}
```

La gestión automática de recursos (también conocida como "try con recurso") refleja el hecho de que las instrucciones try se utilizan a menudo para proteger instrucciones que utilizan recursos que hay que cerrar una vez que se haya terminado de utilizarlos, independientemente de si ese uso ha tenido éxito o ha fallado. La cabecera de la instrucción try se amplía para

incluir la apertura del recurso (que a menudo es un archivo) y el recurso será cerrado automáticamente al final de la instrucción try.

Ejemplo:

```
try (FileWriter writer = new FileWriter(filename)){
    . . .
    Utilizar el escritor. . .
    . . .
}
catch(IOException e) {
    . . .
}
```

D.5 Aserciones

Las *instrucciones de aserción* se utilizan fundamentalmente como herramientas de prueba durante el desarrollo del programa más que en el código de producción. Hay disponibles dos formas de instrucciones de aserción:

```
assert expresión-booleana;
assert expresión-booleana: expresión;
```

Ejemplos:

```
assert getDetails(key) != null;
assert expected == actual:
       "Actual value: " + actual +
       " does not match expected value: " + expected;
```

Si la expresión de aserción se evalúa como *false*, entonces se generará un `AssertionError`.

Una opción del compilador permite desactivar las instrucciones de aserción en el código de producción sin tener que eliminarlas del código fuente del programa.

APÉNDICE

E Ejecución de Java sin BlueJ

A lo largo de este libro, hemos utilizado BlueJ para desarrollar y ejecutar nuestras aplicaciones Java. Hay una buena razón para esto: BlueJ nos proporciona herramientas que facilitan las tareas de desarrollo. En particular, nos permite ejecutar fácilmente métodos de clases individuales; esto resulta muy útil si queremos probar rápidamente un segmento de nuevo código.

Vamos a separarlas explicaciones sobre cómo trabajar sin BlueJ en dos categorías: cómo ejecutar una aplicación sin BlueJ y cómo desarrollar sin BlueJ.

E.1 Ejecución sin BlueJ

Normalmente, cuando se entregan las aplicaciones a los usuarios finales, se las ejecuta de manera diferente que desde dentro de BlueJ. En ese entorno final, las aplicaciones tienen un único punto de entrada, que define dónde empieza la ejecución cuando el usuario arranca la aplicación.

El mecanismo exacto utilizado para iniciar una aplicación depende del sistema operativo. Usualmente, esto se lleva a cabo haciendo doble clic sobre un icono de aplicación o introduciendo el nombre de la aplicación en la línea de comandos. El sistema operativo necesita entonces conocer qué método de qué clase debe invocar para ejecutar el programa completo.

En Java, este problema está ersuleto utilizando un convenio. Cuando se inicia un programa Java, el nombre de la clase se especifica como un parámetro del comando de arranque, y el nombre del método es `main`. El término "main" ha sido elegido arbitrariamente por los desarrolladores de Java, pero es fijo –el método debe tener obligatoriamente este nombre. (La elección de "main" como nombre del método inicial proviene del lenguaje C, del cual Java hereda gran parte de su sintaxis.)

Consideremos, por ejemplo, el siguiente comando introducido en una línea de comandos como el indicativo de comandos de Windows o de un terminal Unix:

```
java Game
```

El comando `java` inicia la máquina virtual Java. Forma parte del *Java Development Kit* (JDK), que debe estar instalado en su sistema. `Game` es el nombre de la clase que deseamos iniciar:

El sistema Java buscará entonces un método en la clase `Game` que tenga exactamente la siguiente signatura:

```
public static void main(String[] args)
```

El método tiene que ser `public`, para que pueda ser invocado desde el exterior. También tiene que ser `static`, porque no existe ningún objeto en el momento de arrancar. Inicialmente

solo tenemos clase, por lo que lo único que podemos invocar son métodos estáticos. A continuación, este método estático se encargará de crear el primer objeto. El tipo de retorno es void, ya que este método no devuelve ningún valor.

El parámetro es una matriz de objetos String. Esto permite a los usuarios pasar argumentos adicionales. En nuestro ejemplo, el valor del parámetro args será una matriz de longitud cero. La línea de comando que arranca el programa puede, sin embargo, definir argumentos:

```
java Game 2 Fred
```

Cada palabra situada después del nombre de la clase en esta línea de comandos será leída como un String separado y será pasada al método main como uno de los elementos de la matriz de cadenas de caracteres. En este caso, la matriz args contendría dos elementos, que son las cadenas de caracteres"2" y "Fred". Con Java no suelen emplearse muy frecuentemente parámetros de la línea de comandos.

El cuerpo del método main puede contener teóricamente cualquier instrucción que deseemos. Sin embargo, las reglas del buen estilo de programación dictan que la longitud del método main debería mantenerse lo más pequeña posible. Específicamente, no debe contener nada que forme parte de la lógica de la aplicación.

Típicamente, el método main debe hacer exactamente lo que nosotros hacíamos de manera interactiva para iniciar la misma aplicación en BlueJ. Por ejemplo, si lo que hacíamos era crear un objeto de la clase Game e invocábamos un método start para iniciar la aplicación, tendríamos que añadir el siguiente método main a la clase Game:

```
public static void main(String[] args)
{
    Game game = new Game();
    game.start();
}
```

Ahora, la ejecución de ese método main simulará nuestra invocación interactiva del juego.

Los proyectos Java suelen almacenarse cada uno en un directorio distinto. Todas las clases de un proyecto se colocan dentro del directorio correspondiente. Cuando ejecute el comando para arrancar Java y ejecute la aplicación, asegúrese de que el directorio de proyectos sea su directorio activo en el terminal de comandos. Esto garantiza que se puedan encontrar las clases.

Si la clase especificada no puede encontrarse, la máquina virtual Java generará un mensaje de error similar al siguiente:

```
Exception in thread "main" java.lang.NoClassDefFoundError: Game
```

Si obtiene un mensaje como este, asegúrese de haber escrito correctamente el nombre de la clase y de que directorio actual contiene realmente esa clase. La clase está almacenada en un archivo con el sufijo .class. Por ejemplo, el código para la clase Game está almacenado en un archivo denominado Game.class.

Si se encuentra la clase pero esta no contiene un método main (o el método main no tiene la signatura correcta), verá un mensaje similar al siguiente:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

En tal caso, asegúrese de que la clase que quiere ejecutar tiene un método main correcto.

E.2 Creación de archivos ejecutables .jar

Los proyectos Java suelen estar almacenados en forma de una colección de archivos en un directorio (o carpeta). Sin embargo, para distribuir aplicaciones a otras personas, a menudo es más sencillo almacenar la aplicación completa en un único archivo. El mecanismo Java para hacer esto es el formato Java Archive (`.jar`). Todos los archivo de una aplicación pueden empaquetarse en un único archivo y continuar siendo ejecutados. (Si está familiarizado con el formato de compresión "zip", puede que le interese saber que el formato es, de hecho, el mismo. Los archivos "Jar" pueden abrirse con programas zip y viceversa.)

Para crear un archivo ejecutable`.jar`, es necesario especificar en algún lugar la clase `main`. (Recuerde: el método que se ejecuta es siempre `main`, pero necesitamos especificar la clase en la que se encuentra este método.) Esto se hace incluyendo en el archivo `.jar` un archivo de texto (el *archivo de manifiesto*) que incluya esa información. Afortunadamente, BlueJ se encarga de hacer eso por nosotros.

Para crear un archivo ejecutable `.jar` en BlueJ, utilice la función *Project—Create Jar File* (Proyecto, Crear archivo jar) y especifique en el cuadro de diálogo que aparece la clase que contiene el método `main`. (Sigue siendo necesario escribir un método `main` de la forma exacta que se ha explicado anteriormente.)

Para conocer más detalles acerca de esta función, lea el tutorial de BlueJ, al que puede acceder a través del menú *Help—Tutorial* de BlueJ o en el sitio web de BlueJ.

Una vez creado el archivo `.jar` ejecutable, puede ejecutarlo haciendo doble clic sobre el mismo. La computadora que ejecute este archivo `.jar` deberá tener instalado el JDK o el JRE (*Java Runtime Environment*) y con él deberán estar asociados los archivo `.jar`.

E.3 Desarrollo sin BlueJ

Si no quiere solo ejecutar, sino también desarrollar sus programas sin BlueJ, tendrá que editar y compilar las clases. El código fuente de una clase se almacena en un archivo que termina en `.java`. Por ejemplo, la clase `Game` está almacenada en un archivo denominado `Game.java`. Los archivos fuente pueden editarse con cualquier editor de texto; en la actualidad hay disponibles muchos editores de texto gratuitos o muy baratos. Algunos de ellos, como el *Bloc de notas* o *WordPad*, se distribuyen con Windows, pero si quiere utilizar el editor para algo más que una prueba rápida de programación, pronto verá que le hace falta conseguir un editor mejor. Tenga cuidado, sin embargo, con los procesadores de texto, porque estos no suelen guardar el texto en formato de texto plano, y el sistema Java no será capaz de leerlo.

Los archivos fuente pueden entonces compilarse desde una línea de comandos utilizando el compilador Java que está incluido en el JDK y que se llama `javac`. Para compilar un archivo fuente denominado `Game.java`, utilice el comando

```
javac Game.java
```

Este comando compilará la clase `Game` y cualesquiera otras clases que dependan de ella, y creará un archivo denominado `Game.class`. Este archivo contiene el código que puede ejecutarse con la máquina virtual Java. Para ejecutarlo, utilice el comando

```
java Game
```

Observe que este comando no incluye el sufijo `.class`.



APÉNDICE F

Utilización del depurador

El depurador de BlueJ proporciona un conjunto de funciones básicas de depuración que están intencionadamente simplificadas, pero que siguen siendo enormemente útiles, tanto para depurar programas como para comprender el comportamiento de esos programas en tiempo de ejecución.

Podemos acceder a la ventana del depurador seleccionando el elemento *Show Debugger* (Mostrar depurador) del menú *View* (Ver) o pulsando con el botón del derecho del ratón sobre el indicador de trabajo y seleccionando *Show Debugger* en el menú emergente. La Figura F.1 muestra la ventana del depurador.



Figura F.1
La ventana del depurador de BlueJ.

La ventana del depurador contiene cinco áreas de visualización y cinco botones de control. Las áreas y los botones se vuelven activos solo cuando un programa alcanza un punto de interrupción o se detiene por alguna otra razón. En las siguientes secciones se describe cómo establecer puntos de interrupción, cómo controlar la ejecución del programa y el propósito de cada una de las áreas de visualización.

F.1 Puntos de interrupción

Un punto de interrupción es un indicador asociado a una línea de código (Figura F.2). Cuando se alcanza un punto de interrupción durante la ejecución del programa, las áreas de visualización y los controles del depurador se activan, permitiéndonos inspeccionar el estado del programa y controlar su ejecución.

Figura F.2
Un punto de interrupción asociado a una línea de código.

```
/**
 * Print the next mail item (if any) for this user to the text
 * terminal.
 */
public void printNextMailItem()
{
    MailItem item = server.getNextMailItem(user);
    if(item == null) {
        System.out.println("No new mail.");
    }
    else {
        item.print();
    }
}
```

Los puntos de interrupción se definen mediante la ventana del editor. Pulse con el botón izquierdo del ratón en el área de puntos de interrupción situada a la izquierda del código fuente o coloque el cursor en la línea de código en la que quiera definir el punto de interrupción y seleccione *Set/Clear Breakpoint* (Establecer/Eliminar punto de interrupción) en el menú *Tools* del editor. Los puntos de interrupción pueden eliminarse siguiendo el proceso inverso. Los puntos de interrupción solo se pueden definir dentro de las clases que ya hayan sido compiladas.

F.2 Los botones de control

La Figura F.3 muestra los botones de control que están activos en un punto de interrupción.

Figura F.3
Botones de control activos en los puntos de interrupción.



F.2.1 *Halt*

El botón *Halt* (Detener) está activo cuando el programa se está ejecutando, permitiendo así interrumpir la ejecución en caso necesario. Si se detiene la ejecución, el depurador mostrará el estado del programa exactamente igual que si se hubiera alcanzado un punto de interrupción.

F.2.2 *Step*

El botón *Step* (Dar un paso) reanuda la ejecución en la instrucción actual. La ejecución volverá a detenerse una vez completada esa instrucción. Si la instrucción implica una llamada a método,

esta se completa antes de que la ejecución vuelva a detenerse (a menos que esa llamada nos conduzca a otro punto de interrupción explícito).

F.2.3 *Step Into*

El botón *Step Into* (Dar un paso al interior) reanuda la ejecución en la instrucción actual. Si esta instrucción es una llamada a método, entonces la ejecución saltará al interior (*step into*) de dicho método y se detendrá en la primera instrucción del mismo.

F.2.4 *Continue*

El botón *Continue* (Continuar) reanuda la ejecución hasta alcanzar el siguiente punto de interrupción, hasta que se interrumpa la ejecución mediante el botón *Halt* o hasta que la ejecución se complete normalmente.

F.2.5 *Terminate*

El botón *Terminate* (Terminar) finaliza de manera agresiva la ejecución del programa actual, de modo que no puede ya reanudarse de nuevo. Si lo único que queremos es interrumpir la ejecución para examinar el estado actual del programa, entonces es preferible utilizar la operación *Halt*.

F.3 Las áreas de visualización de variables

La Figura F.4 muestra las tres áreas de visualización de variables que están activas en un punto de interrupción y muestran un ejemplo de la simulación predadores-presas que hemos visto en el Capítulo 10. Las variables estáticas se muestran en el área superior, las variables de instancia en el área central y las variables locales en el área inferior.

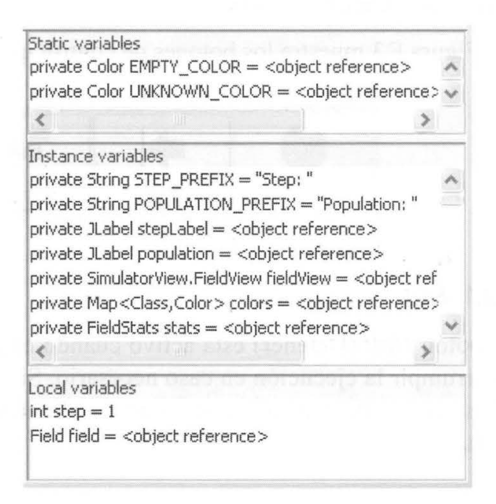


Figura F.4
Áreas de visualización de variables activas.

Cuando se alcance un punto de interrupción, la ejecución se detendrá en una instrucción correspondiente a un objeto arbitrario dentro del programa actual. El área *Static variables* muestra los valores de las variables estáticas definidas en la clase de dicho objeto. El área *Instance*

variables muestra los valores de las variables de instancia de ese objeto concreto. Ambas áreas incluyen también todas las variables heredadas de las superclases.

El área *Local variables* muestra los valores de las variables locales y parámetros del constructor o método que se esté ejecutando actualmente. Las variables locales aparecerán en esta área solo después de haber sido inicializadas, ya que solo entonces pasan a existir esas variables dentro de la máquina virtual.

Cualquier variable que aparezca en una de estas áreas y que sea una referencia a objeto puede ser inspeccionada haciendo doble clic sobre ella.

F.4 El área Call Sequence

La Figura F.5 muestra el área de visualización *Call Sequence*, que contiene una secuencia de llamadas de seis métodos de profundidad. Los métodos aparecen en esa secuencia en el formato `Class.method`, independientemente de si se trata de métodos estáticos o de métodos de instancia. Los constructores aparecen en la secuencia con el formato `Class.<init>`.

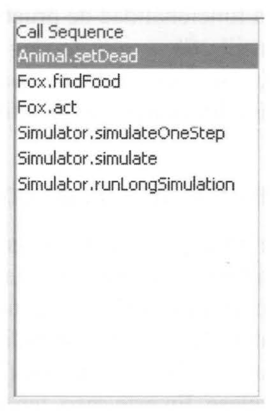


Figura F.5
Una secuencia de llamadas.

La secuencia de llamadas opera como una pila, siendo el método que aparece en la parte superior de la secuencia aquel en el que se encuentra actualmente el flujo de ejecución. Las áreas de visualización de variables reflejan los detalles del método o constructor que aparezca actualmente resaltado dentro de la secuencia de llamadas. Si seleccionamos una línea distinta de la secuencia, se actualizará el contenido de las otras áreas de visualización.

F.5 El área de visualización Threads

El área de visualización *Threads* cae fuera del alcance de este libro y no hablaremos acerca de ella.

APÉNDICE

G Herramientas JUnit de prueba de unidades

En este apéndice, vamos a realizar un breve esbozo de las principales funciones que BlueJ ofrece para dar soporte a la prueba de unidades estilo JUnit. Puede encontrar más detalles en el tutorial sobre pruebas disponible en el CD del libro y en el sitio web de BlueJ.

G.1 Activación de la funcionalidad de prueba de unidades

Para activar la funcionalidad de prueba de unidades de BlueJ, es necesario asegurarse de tener marcada la casilla *Show unit testing tools* (Mostrar herramientas de prueba de unidades) en el menú *Tools-Preferences-Miscellaneous*. La ventana principal de BlueJ contendrá entonces varios botones adicionales que se activarán cuando se abra un proyecto.

G.2 Creación de una clase de prueba

Las clases de prueba se crean haciendo clic con el botón derecho del ratón sobre una clase de las que aparecen en el diagrama de clases y seleccionando *Create Test Class* (Crear clase de prueba). El nombre de la clase de prueba se determina automáticamente añadiendo `Test` como sufijo al nombre de la clase asociada. Alternativamente, se puede crear una clase de prueba seleccionando el botón *New Class* (Nueva clase) y eligiendo *Unit Test* (Prueba de unidad) como tipo de clase. En este caso, tenemos completa libertad para definir el nombre.

Las clases de prueba están anotadas con la indicación `<<unit test>>` en el diagrama de clases y tienen un color distinto al de las clases normales.

G.3 Creación de un método de prueba

Los métodos de prueba se pueden crear interactivamente. Se graba una secuencia de interacciones del usuario con el diagrama de clases y con el banco de objetos, y luego se captura esa secuencia en forma de una serie de declaraciones e instrucciones Java dentro de un método de la clase de prueba. Podemos iniciar la grabación seleccionando *Create Test Method* (Crear método de prueba) en el menú emergente asociado con una clase de prueba. Se nos pedirá que introduzcamos el nombre del nuevo método. En las primeras versiones de JUnit, hasta la versión 3, se requería que los nombres de método comenzarán con el prefijo "test"; sin embargo, esto ya no es necesario en la versiones actuales. El símbolo de *grabación* situado a la izquierda del diagrama de clases aparecerá entonces con un color rojo y los botones *End* y *Cancel* pasarán a estar disponibles.

Una vez que se ha iniciado la grabación, cualquier creación de objeto o llamada a método pasará a formar parte del código del método que se está creando. Seleccione *End* para terminar

la grabación y capturar la prueba, o seleccione *Cancel* para descartar la grabación, sin hacer ninguna modificación en la clase de prueba.

Los métodos de prueba tienen la anotación `@Test` en el código fuente de la clase de prueba.

G.4 Aserciones de prueba

Mientras se graba un método de prueba, toda llamada a método que devuelva un resultado hará que aparezca una ventana *Method Result* (Resultados del método). Esta ventana ofrece la oportunidad de definir una aserción acerca del valor obtenido como resultado, seleccionando la casilla *Assert that* (incluir aserción). Un menú desplegable ofrece un conjunto de posibles aserciones para el resultado. Si se hace una aserción, esta será codificada dentro del método de prueba en forma de una llamada a método, cuyo objeto es provocar la generación de un error `AssertionError` si la prueba falla.

G.5 Ejecución de las pruebas

Pueden ejecutarse métodos de prueba individuales seleccionándolos en el menú emergente asociado con la clase de prueba. La correcta terminación de un prueba se indicará mediante un mensaje en la línea de estado de la ventana principal. Una prueba fallida hará que aparezca la ventana *Test Results* (Resultados de la prueba). Si seleccionamos *Test All* (Probar todo) en el menú emergente de la clase de prueba se ejecutarán todas las pruebas de esa clase de prueba individual. La ventana *Test Results* nos detallará los métodos que han tenido éxito y los que han fallado.

G.6 Bancos de pruebas

El contenido del banco de objetos puede ser capturado en forma de *banco de pruebas* (*fixture*) seleccionando *Object Bench to Test Fixture* (Del banco de objetos al banco de pruebas) en el menú emergente asociado con la clase de pruebas. El efecto de crear un banco de pruebas es que se añade a la clase de pruebas una definición de campo por cada objeto, y se añaden también a su método `setUp` una serie de instrucciones que permiten volver a crear el estado exacto de los objetos, tal como aparecen en el banco de objetos. A continuación, los objetos se eliminan del banco de objetos.

El método `setUp` tiene la anotación `@Before` en la clase de prueba y ese método se ejecuta automáticamente antes de ejecutar cualquier método de prueba, para que de este modo estén disponibles para todas las pruebas todos los objetos del banco de pruebas.

Los objetos de un banco de pruebas pueden volver a ser creados en el banco de objetos seleccionando *Test Fixture to Object Bench* (Del banco de pruebas al banco de objetos) en el menú de la clase de pruebas.

Después de cada método de prueba se invoca un método `tearDown`, con la anotación `@After`. Este método puede emplearse para realizar cualquier tarea que sea necesaria después de las pruebas o para llevar a cabo operaciones de limpieza, en caso necesario.

APÉNDICE

H Herramientas para trabajo en equipo

En este apéndice vamos a describir brevemente las herramientas disponibles para soporte al trabajo en equipo.

BlueJ incluye herramientas de trabajo en equipo basadas en un modelo de repositorio de código fuente. En este modelo, se configura un servidor repositorio al que se puede acceder a través de Internet desde las máquinas en las que estén trabajando los usuarios.

Es necesario que un administrador configure el servidor. BlueJ soporta tanto repositorios Subversion como CVS.

H.1 Configuración del servidor

La configuración del servidor repositorio debe ser realizada, normalmente, por un administrador con experiencia. Puede encontrar instrucciones detalladas en el CD del libro dentro del documento titulado *repository-setup.pdf.*

H.2 Activación de la funcionalidad de trabajo en equipo

Las herramientas de trabajo en equipo están inicialmente ocultas en BlueJ. Para visualizarlas, abra el cuadro de diálogo *Preferences* y, en la pestaña *Miscellaneous*, active la casilla *Show teamwork controls* (Mostrar controles de trabajo en equipo). La interfaz de BlueJ contendrá entonces tres botones adicionales: *Update* (Actualización), *Commit* (Confirmación), *Status* (Estado) y un submenú adicional denominado *Team* (Equipo) en el menú *Tools*.

H.3 Compartición de un proyecto

Para crear un proyecto compartido, uno de los miembros del equipo crea el proyecto en forma de proyecto BlueJ estándar. El proyecto puede compartirse entonces utilizando la función *Share this Project* (Compartir este proyecto) del menú *Team*. Cuando se emplea esta función, se almacena en el repositorio central una copia del proyecto. Será necesario especificar el nombre del servidor y los detalles de acceso en un cuadro de diálogo; pregunte a su administrador (al que haya configurado el repositorio) cuáles son los detalles que hay que facilitar en ese cuadro de diálogo.

H.4 Utilización de un proyecto compartido

Una vez que un usuario ha creado un proyecto compartido en el repositorio, otros miembros del equipo pueden usarlo. Para ello, seleccione *Checkout Project* (Acceder al proyecto) en el

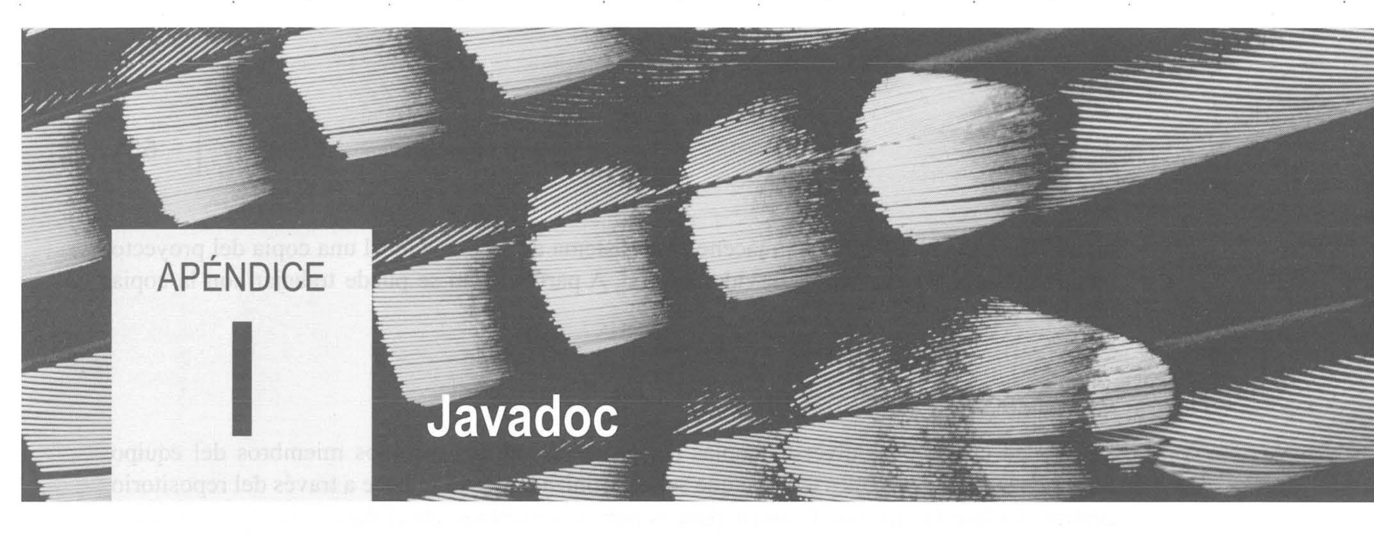
menú *Team*. Esto hará que se almacene en el sistema de archivos local una copia del proyecto compartido almacenado en el servidor central. A partir de ahí se puede trabajar con la copia local.

H.5 Actualización y confirmación

De vez en cuando, las diversas copias del proyecto que los distintos miembros del equipo tienen en sus discos locales tendrán que ser sincronizadas. Esto se hace a través del repositorio central. Utilice la función *Commit* para copiar sus cambios en el repositorio y la función *Update* para copiar en su propia copia local cambios del repositorio (cambios que otros miembros del equipo hayan confirmado). Es aconsejable confirmar y actualizar los cambios frecuentemente, con el fin de que los cambios realizados en cualquier momento no sean demasiado sustanciales.

H.6 Más información

Hay disponible información más detallada en el documento *Team Work Tutorial*, que se encuentra en el CD del libro con el nombre *teamwork-tutorial.pdf*.



APÉNDICE I Javadoc

El escribir una buena documentación para las definiciones de clases e interfaces es un importante complemento a la escritura de un código fuente de buena calidad. La documentación nos permite comunicar nuestras intenciones a las personas que lean el código fuente, proporcionando una panorámica de alto nivel en lenguaje natural, en lugar de forzarlas a leer código fuente de relativo bajo nivel. De particular importancia es la documentación para los elementos `public` de una clase o interfaz, de modo que los programadores puedan hacer uso de la misma sin tener que conocer los detalles de su implementación.

En todos los ejemplos de proyecto de este libro, hemos utilizado un estilo de comentarios concreto, que es el reconocido por la herramienta de documentación `javadoc` y que se distribuye como parte del JDK. Esta herramientas automatiza la generación de la documentación de una clase en forma de páginas HTML con un estilo coherente. La API Java se ha documentado utilizando esta misma herramienta y puede apreciarse perfectamente el gran valor que tiene cuando se utilizan clases de librería.

En este apéndice, proporcionamos un breve resumen de los elementos principales de los comentarios de documentación que todo programador debería habituarse a utilizar en su propio código fuente.

I.1 Comentarios de documentación

Los elementos de una clase que hay que documentar son la definición de la clase como un todo, sus campos, sus constructores y sus métodos. Lo más importante desde el punto de vista de un usuario de nuestras clases es disponer de documentación sobre la clase y sus constructores y métodos públicos. Nosotros hemos tendido a no proporcionar estilos javadoc para los campos, porque consideramos que estos son detalles privados del nivel de implementación, por lo que no se trata de información que los usuarios deban manejar.

Los comentarios de documentación se abren siempre con los tres caracteres "/**" y se cierran con la pareja de caracteres "*/". Entre estos símbolos, el comentario tendrá una *descripción principal* seguida de una *sección de marcadores*, aunque ambos elementos son opcionales.

I.1.1 *La descripción principal*

La descripción principal de una clase debería ser una descripción general del propósito de esa clase. El Código I.1 muestra parte de una descripción principal típica, tomada de la clase `Game` del proyecto *world-of-zuul*. Observe que la descripción incluye detalles de cómo utilizar esta clase para iniciar el juego.

Código I.1
La descripción principal en el comentario de una clase.

```
/**
 * Esta clase es la clase principal de de la aplicación
 * "World of Zuul". Esta aplicación es un juego de aventuras
 * muy simple basado en texto.
 * Los usuarios se pueden desplazar por varias salas y eso es todo.
 * ¡Debería ampliarse para hacerlo más interesante!
 * Para jugar, cree una instancia de esta clase e invoque
 * el método "play" .
 */
```

La descripción principal de un método debería ser bastante general, sin entrar en demasiados detalles acerca de cómo se implementa el método. De hecho, la descripción principal de un método no necesitará tener más de una sola frase, como por ejemplo

```
/**
 * Crear un nuevo pasajero con lugares distintos de
 * recogida y de destino.
 */
```

Hay que pensar con especial cuidado la primera frase de la descripción principal de una clase, interfaz o método, ya que esa frase se utiliza para elaborar un resumen independiente, que aparece al principio de la documentación generada.

Javadoc también admite el uso de marcadores HTML dentro de estos comentarios.

I.1.2 *La sección de marcadores*

A la descripción principal le sigue la *sección de marcadores*. Javadoc reconoce unos 20 marcadores distintos, aunque aquí solo vamos a hablar de los más importantes (Tabla I.1). Los marcadores pueden utilizarse de dos formas: *marcadores de bloque* y *marcadores incrustados*. Solo vamos a ocuparnos de los marcadores de bloque, ya que son los más comúnmente utilizados. En la sección *javadoc* de la documentación *Tools and Utilities*, que forma parte del JDK, podrá encontrar más detalles sobre los marcadores incrustados y los restantes marcadores disponibles.

Tabla I.1
Marcadores `javadoc` comunes.

Marcador	**Texto asociado**
`@author`	Nombre del autor o autores.
`@param`	Nombre y descripción del parámetro.
`@return`	Descripción del valor de retorno.
`@see`	Referencia cruzada.
`@throws`	Tipo de excepción generado y las circunstancias.
`@version`	Descripción de la versión.

Los marcadores `@author` y `@version` se suelen encontrar normalmente en los comentarios de clases y de interfaces, y no pueden emplearse en los comentarios de constructores, métodos

o campos. Ambos marcadores van seguidos por un texto libre, no existiendo ningún formato obligatorio en ninguno de los dos casos. He aquí un ejemplo:

```
@author Hacker T. Largebrain
@version 2012.12.03
```

Los marcadores `@param` y `@throws` se utilizan con métodos y constructores, mientras que `@return` se usa solo con métodos. He aquí algunos ejemplos:

```
@param limit El valor máximo permitido.
@return Un número aleatorio en el rango de 1 al límite (inclusive).
@throws IllegalLimitException Si el límite es menor que 1.
```

El marcador `@see` dispone de varios formatos y puede utilizarse en cualquier comentario de documentación. Proporciona una forma de establecer una referencia cruzada entre el comentario y alguna otra clase o método, o alguna otra forma de documentación. Se añadirá una sección *See Also* al elemento que se está comentando. He aquí algunos ejemplos típicos:

```
@see "The Java Language Specification, by Joy et al"
@see <a href="http://www.bluej.org/">The BlueJ web site</a>
@see #isAlive
@see java.util.ArrayList#add
```

El primer marcador simplemente incrusta una cadena de texto sin ningún hipervínculo; el segundo incrusta un hipervínculo al documento especificado. El tercero enlaza con la documentación del método `isAlive` de la misma clase; el cuarto enlaza con la documentación del método `add` de la clase `ArrayList` del paquete `java.util`.

I.2 Soporte de BlueJ para javadoc

Si se ha comentado un proyecto utilizando el estilo *javadoc*, entonces BlueJ proporciona soporte para generar la documentación HTML completa. En la ventana principal, seleccione el elemento de menú *Tools/Project Documentation* y eso hará que se genere la documentación (si es necesario) y que se muestre en la ventana de un explorador.

Dentro del editor de BlueJ, puede alternarse entre la vista del código fuente de una clase y la vista de documentación cambiando la opción *Source Code* (Código fuente) a *Documentation* en la parte derecha de la ventana (Figura I.1) o utilizando la opción *Toggle Documentation View* (Cambiar a vista documentación) del menú *Tools* del editor. Esto proporciona una previsualización rápida de la documentación, pero no contendrá referencias a la documentación de las superclases o a las clases utilizadas.

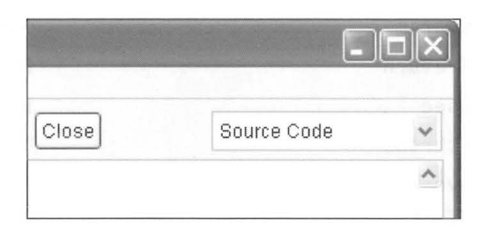


Figura I.1 Opción de visualización de *Source Code* y *Documentation*.

Puede encontrar más información en: `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`



J.1 Denominación

J.1.1 *Utilice nombres significativos*

Utilice nombres descriptivos para todos los identificadores (nombres de clases, variables y métodos). Evite la ambigüedad, evite la abreviaciones. Los métodos mutadores simples deberían denominarse *setLoquesea(...)*. Los métodos selectores simples deben denominarse *getLoquesea(...)*. Los métodos selectores con valores de retorno de tipo `boolean` a menudo se denominan *isLoquesea(...)*, como por ejemplo `isEmpty()`.

J.1.2 *Los nombres de clase comienzan con una letra mayúscula*

J.1.3 *Los nombres de clase son nombres en singular*

J.1.4 *Los nombres de métodos y variables empiezan con letras en minúscula*

Los tres tipos de nombres (de clases, de métodos y de variables) utilizan letras mayúsculas entremedias para incrementar la legibilidad de los identificadores compuestos, como por ejemplo en `numberOfItems`.

J.1.5 *Las constantes se escriben en MAYÚSCULAS*

Las constantes emplean ocasionalmente guiones bajos para indicar identificadores compuestos, como `MAXIMUM_SIZE`.

J.2 Disposición del texto

J.2.1 *Cada de nivel de sangrado equivale a cuatro espacios*

J.2.2 *Todas las instrucciones dentro de un bloque se sangran un nivel*

J.2.3 *Las llaves para las clases y métodos se escriben solas en una línea*

Las llaves para los bloques de clases y métodos se escriben en líneas separadas y tienen el mismo nivel de sangrado. Por ejemplo:

```
public int getAge()
{
    instrucciones
}
```

J.2.4 *Para todos los demás bloques, las llaves se abren al final de una línea*

Los demás bloques se abren con llaves situadas al final de la línea que contiene la palabra clave que define el bloque. La llave de cierre se incluye en una línea separada, alineada con la palabra clave que define el bloque. Por ejemplo:

```
while(condición) {
    instrucciones
}

if(condición) {
    instrucciones
}
else {
    instrucciones
}
```

J.2.5 *Utilice siempre llaves en las estructuras de control*

Se utilizan llaves en las instrucciones if y en los bucles, incluso aunque el cuerpo solo contenga una instrucción.

J.2.6 *Utilice un espacio antes de la llave de apertura de un bloque de una estructura de control*

J.2.7 *Utilice un espacio alrededor de los operadores*

J.2.8 *Utilice una línea en blanco entre métodos (y constructores)*

Utilice líneas en blanco para separar bloques lógicos de código; esto significa que deben emplearse al menos entre un método y otro, pero también entre las distintas partes lógicas contenidas dentro de un método.

J.3 Documentación

J.3.1 *Toda clase debe tener un comentario de clase al principio*

El comentario de la clase contendrá como mínimo:

- Una descripción general de la clase.
- El nombre del autor o autores.
- Un número de versión.

Toda persona que haya contribuido a elaborar esa clase debe ser incluida como autor, o debe de algún otro modo reconocerse su contribución.

El número de versión puede ser un simple número, una fecha o tener cualquier otro formato. Lo importante es que el lector pueda reconocer si dos versiones son diferentes y determinar cuál de ellas es más reciente.

J.3.2 *Todo método tiene un comentario de método*

J.3.3 *Los comentarios son legibles por Javadoc*

Los comentarios de clases y métodos deben ser reconocidos por Javadoc. En otras palabras, deben comenzar con el símbolo de comentario "/**".

J.3.4 *Deben utilizarse comentarios de código (solo) cuando sea necesario*

Deben incluirse comentarios en el código cuando este no sea obvio o resulte difícil de entender (debiendo darse preferencia a la escritura de código que sea obvio o fácil de entender, siempre que sea posible) y allí donde los comentarios faciliten la comprensión de un método. No comente instrucciones obvias, ¡presuponga que su lector comprende Java!

J.4 Restricciones en el uso del lenguaje

J.4.1 *Orden de las declaraciones: campos, constructores, métodos*

Los elementos de la definición de una clase aparecen (si están presentes) en el siguiente orden: instrucción de paquete; instrucciones de importación; comentario de clase; cabecera de clase; definiciones de campos; constructores; métodos; clases internas.

J.4.2 *Los campos no deben ser públicos (excepto los de tipo final)*

J.4.3 *Utilice siempre un modificador de acceso*

Especifique todos los campos y métodos como `private`, `public` o `protected`. No utilice nunca el acceso predeterminado (privado de nivel de paquete).

J.4.4 *Importe las clases por separado*

Es preferible utilizar instrucciones de importación que identifiquen explícitamente cada clase, en lugar de importar paquetes completos. Por ejemplo,

```
import java.util.ArrayList;
import java.util.HashSet;
```

es mejor que

```
import java.util.*;
```

J.4.5 *Incluya siempre un constructor (incluso si el cuerpo está vacío)*

J.4.6 *Incluya siempre una llamada al constructor de la superclase*

En los constructores de las subclases, no confíe en la inserción automática de una llamada a la superclase. Incluya explícitamente la llamada a `super()`, incluso aunque el código funcione sin necesidad de incluirla.

J.4.7 *Inicialice todos los campos en el constructor*

J.5 Normas para el código

J.5.1 *Utilice iteradores con las colecciones*

Para iterar a través de una colección completa, utilice un bucle for-each. Cuando sea necesario modificar la colección durante la iteración, utilice un `Iterator` con un bucle while o un bucle for, no un índice entero.



La plataforma Java incluye un rico conjunto de librerías que dan soporte a una amplia variedad de tareas de programación.

En este apéndice resumimos brevemente los detalles de algunas clases e interfaces de los paquetes más importantes de la API Java. Todo programador Java competente debería estar familiarizado con la mayoría de estas. Este apéndice es solo un resumen y debe ser leído junto con la documentación completa de la API.

K.1 El paquete java.lang

Las clases e interfaces del paquete `java.lang` son fundamentales para el lenguaje Java, ya que este paquete se importa implícitamente de manera automática en cualquier definición de clase.

paquete `java.lang` – Resumen de las clases más importantes	
interfaz `Comparable`	La implementación de esta interfaz permite la comparación y ordenación de objetos de la clase implementadora. Una serie de métodos estáticos de utilidad como `Arrays.sort` y `Collections.sort` pueden entonces proporcionar un mecanismo eficiente de ordenación en tales casos, por ejemplo.
clase `Math`	`Math` es una clase que solo contiene métodos y campos estáticos. Los valores de las constantes matemáticas e y π se definen aquí, junto con funciones trigonométricas y otras como `abs`, `min`, `max` y `sqrt`.
clase `Object`	Todas las clases tienen `Object` como superclase en la raíz de su jerarquía de clases. De dicha clase raíz, todos los objetos heredan implementaciones predeterminadas para una serie de métodos importantes como `equals` y `toString`. Otros métodos relevantes definidos por esta clase son `clone` y `hashCode`.
clase `String`	Las cadenas de caracteres son una característica importante de muchas aplicaciones y reciben un tratamiento especial en Java. Los métodos clave de la clase `String` son `charAt`, `equals`, `indexOf`, `length`, `split` y `substring`. La cadenas de caracteres son objetos inmutables, por lo que métodos como `trim` que parecen ser mutadores, lo que devuelven en realidad es un nuevo objeto `String` que representa el resultado de la operación.
clase `StringBuilder`	La clase `StringBuilder` ofrece una alternativa eficiente a `String` cuando lo que se necesita es construir una cadena a partir de una serie de componentes: por ejemplo, mediante concatenación. Sus métodos principales son `append`, `insert` y `toString`.

K.2 El paquete java.util

El paquete `java.util` es una colección relativamente incoherente de clases e interfaces útiles.

paquete `java.util` – Resumen de las clases e interfaces más importantes	
interfaz `Collection`	Esta interfaz proporciona el conjunto fundamental de métodos para la mayoría de las clases basadas en colección definidas en el paquete `java.util`, como `ArrayList`, `HashSet` y `LinkedList`. Define signaturas para los métodos `add`, `clear`, `iterator`, `remove` y `size`.
interfaz `Iterator`	`Iterator` define una interfaz simple y coherente para iterar a través del contenido de una colección. Sus tres métodos son: `hasNext`, `next` y `remove`.
interfaz `List`	`List` es una extensión de la interfaz `Collection` y proporciona un medio de imponer una secuencia a la selección. Como tal, muchos de sus métodos admiten un parámetro de índice: por ejemplo `add`, `get`, `remove` y `set`. Clases como `ArrayList` y `LinkedList` implementan `List`.
interfaz `Map`	La interfaz `Map` ofrece una alternativa a las colecciones basadas en lista, dando soporte a la idea de asociar cada objeto de una colección con un valor *clave*. Los objetos se añaden y se extraen a través de sus métodos `put` y `get`. Observe que un `Map` no devuelve un `Iterator`, sino que su método `keySet` devuelve un `Set` de las claves y su método `values` devuelve una `Collection` de los objetos del mapa.
interfaz `Set`	`Set` amplía la interfaz `Collection` con la intención de obligar a que una colección no contenga elementos duplicados. Merece la pena resaltar que, debido a que es una interfaz, `Set` no tiene ninguna capacidad real de imponer esta restricción. Esto significa que `Set` se proporciona en realidad como interfaz indicadora, para permitir a los implementadores de una colección que le indiquen que sus clases satisfacen esta restricción concreta.
clase `ArrayList`	`ArrayList` es una implementación de la interfaz `List` que utiliza una matriz para proporcionar un acceso directo y eficiente, a través de índices enteros, a los objetos que almacena. Si se añaden o eliminan objetos de cualquier posición de la lista que no sea la última, los siguientes elementos tendrán que ser movidos para hacer espacio o cerrar el hueco. Los métodos fundamentales son: `add`, `get`, `iterator`, `remove` y `size`.
clase `Collections`	`Collections` contiene muchos métodos estáticos útiles para manipular colecciones. Sus métodos fundamentales son: `binarySearch`, `fill` y `sort`.
clase `HashMap`	`HashMap` es una implementación de la interfaz `Map`. Los métodos fundamentales son `get`, `put`, `remove` y `size`. La iteración a través de un `HashMap` suele ser un proceso en dos etapas: se obtiene el conjunto de claves mediante su método `keySet` y luego se itera a través de las claves.
clase `HashSet`	`HashSet` es una implementación de tipo hash de la interfaz `Set`. Su uso es más parecido al de una `Collection` que al de un `HashMap`. Sus métodos fundamentales: son `add`, `remove` y `size`.

Continúa

paquete `java.util` – Resumen de las clases e interfaces más importantes	
clase `LinkedList`	`LinkedList` es una implementación de la interfaz `List` que utiliza una estructura interna enlazada para almacenar los objetos. El acceso directo a los extremos de la lista es eficiente, pero el acceso a los objetos individuales a través de un índice es menos eficiente que con un `ArrayList`. Por otro lado, añadir o eliminar objetos de la lista no requiere desplazar los objetos existentes. Sus métodos fundamentales son: `add`, `getFirst`, `getLast`, `iterator`, `removeFirst`, `removeLast` y `size`.
clase `Random`	La clase `Random` soporta la generación de valores pseudoaleatorios, normalmente números aleatorios. La secuencia de números generada se determina mediante un *valor semilla*, que puede pasarse a un constructor o puede definirse mediante una llamada a `setSeed`. Dos objetos `Random` que comiencen a partir de la misma semilla devolverán la misma secuencia de valores a las sucesivas llamadas. Sus métodos fundamentales son: `nextBoolean`, `nextDouble`, `nextInt` y `setSeed`.
clase `Scanner`	La clase `Scanner` proporciona una forma de leer y analizar sintácticamente la entrada. A menudo se utiliza para leer la entrada procedente de un teclado. Sus métodos fundamentales son: `next` y `hasNext`.

K.3 Los paquetes java.io y java.nio.file

Los paquetes `java.io` y `java.nio.file` contienen clases que dan soporte a la entrada, la salida y el acceso al sistema de archivos. Muchas de las clases de entrada/salida de `java.io` se pueden clasificar según sean *basadas en flujos* (es decir, que operan con datos binarios) o *lectores* y *escritores* (que operan con caracteres). El paquete `java.nio.file` suministra varias clases que dan soporte y un cómodo acceso al sistema de archivos.

paquete `java.io.file` – Resumen de las clases e interfaces más importantes	
interfaz `Serializable`	La interfaz `Serializable` es una interfaz vacía que no requiere que se escriba ningún código en las clases que la implementen. Las clases implementan esta interfaz para poder participar en el proceso de serialización. Los objetos `Serializable` pueden ser escritos y leídos como un todo en y desde los orígenes de salida y de entrada. Esto hace que el almacenamiento y extracción de datos persistentes sea un proceso relativamente simple en Java. Para obtener más información, vea las clases `ObjectInputStream` y `ObjectOutputStream`.
clase `BufferedReader`	`BufferedReader` es una clase que proporciona acceso con buffer basado en caracteres a un origen de entrada. La entrada con buffer suele ser más eficiente que la que no tiene buffer, particularmente si el origen de entrada se encuentra en el sistema de archivos externo. Puesto que almacena la entrada con buffer, es capaz de ofrecer un método `readLine` que no está disponible en la mayor parte de las restantes clases de entrada. Sus métodos fundamentales son: `close`, `read` y `readLine`.

Continúa

paquete `java.io.file` – Resumen de las clases e interfaces más importantes	
class `BufferedWriter`	`BufferedWriter` es una clase que proporciona salida basada en caracteres con buffer. La salida con buffer suele ser más eficiente que la que no tiene buffer, particularmente si el destino de la salida es el sistema de archivos externo. Sus métodos fundamentales son: `close`, `flush` y `write`.
class `File`	La clase `File` proporciona una representación en forma objeto para los archivos y carpetas (directorios) en un sistema de archivos externo. Hay métodos para indicar si se puede leer y/o escribir en un archivo, así como para indicar si se trata de un archivo o de una carpeta. Puede crearse un objeto `File` para un archivo no existente, por ejemplo como primer paso para la creación de un archivo físico en el sistema de archivos. Sus métodos fundamentales son: `canRead`, `canWrite`, `createNewFile`, `createTempFile`, `getName`, `getParent`, `getPath`, `isDirectory`, `isFile` y `listFiles`.
class `FileReader`	La clase `FileReader` se utiliza para abrir un archivo externo y dejarlo listo para leer su contenido en forma de caracteres. Los objetos `FileReader` se suelen a pasar a menudo al constructor de otra clase lectora (como por ejemplo `BufferedReader`), en lugar de utilizarlos directamente. Sus métodos fundamentales: son `close` y `read`.
clase `FileWriter`	La clase `FileWriter` se utiliza para abrir un archivo externo y dejarlo listo para escribir en él datos basados en caracteres. Existen parejas de constructores que determinan si se van a añadir los datos a un archivo existente o si el contenido existente debe ser descartado. Los objetos `FileWriter` suelen pasarse al constructor de otra clase `Writer` (como por ejemplo `BufferedWriter`), en lugar de utilizarse directamente. Sus métodos fundamentales son: `close`, `flush` y `write`.
clase `IOException`	`IOException` es una clase de excepción comprobada que se encuentra en la raíz de la jerarquía de excepciones de la mayoría de las excepciones de entrada/salida.
interfaz `Path`	La interfaz `Path` proporciona los métodos fundamentales para acceder a información acerca de un archivo almacenado en un sistema de archivos. `Path` es, de hecho, un sustituto de la antigua clase `File` del paquete `java.io`.
clase `Paths`	La clase `Paths` proporciona métodos `get` para devolver instancias concretas de la interfaz `Path`.
clase `Files`	`Files` es una clase que proporciona métodos estáticos para consultar atributos de los archivos y directorios (carpetas), así como para manipular el sistema de archivos –por ejemplo, crear directorios y modificar los permisos de archivo. También incluye métodos para abrir archivos, como `newBufferedReader`.

K.4 El paquete java.net

El paquete `java.net` contiene clases e interfaces para dar soporte a las aplicaciones en red. La mayoría de ellas caen fuera del alcance de este libro.

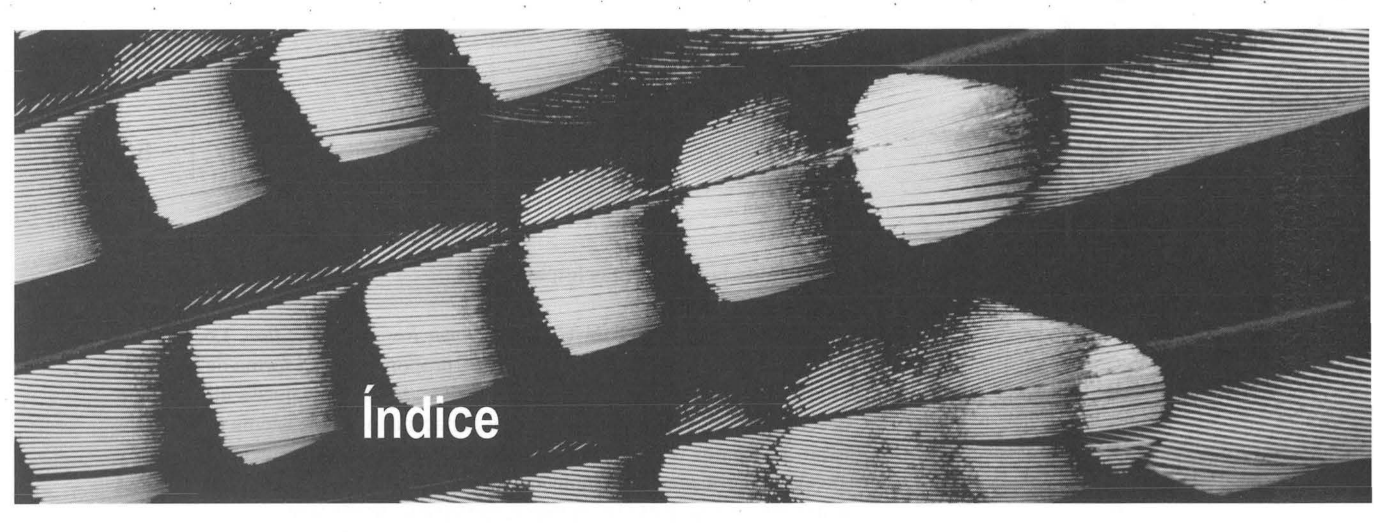
paquete `java.net` – Resumen de las clases más importantes	
clase `URL`	La clase `URL` representa un URL (*Uniform Resource Locator*, Localizador uniforme de recursos). En otras palabras, proporciona una forma de describir algo que se encuentra en Internet. De hecho, también se puede utilizar para describir la ubicación de algo que esté almacenado en un sistema de archivos local. La hemos incluido aquí porque las clases de los paquetes `java.io` y `javax.swing` a menudo emplean objetos `URL`. Sus métodos fundamentales son: `getContent`, `getFile`, `getHost`, `getPath` y `openStream`.

K.5 Otros paquetes importantes

Otros paquetes importantes son:

```
java.awt
java.awt.event
javax.swing
javax.swing.event
```

Estos se usan ampliamente a la hora de escribir interfaces gráficas de usuario (GUI) y contienen muchas clases útiles con las que todo programador de interfaces GUI debería estar familiarizado.



Índice

Contrato de Licencia de Código Binario de Oracle para los productos de la plataforma Java SE Platform

ORACLE AMERICA, INC. (EN ADELANTE DENOMINADO "ORACLE") LE CONCEDE, EN SU PROPIO NOMBRE Y EN EL DE SUS FILIALES, LA LICENCIA DEL SOFTWARE ÚNICAMENTE CON LA CONDICIÓN DE QUE USTED ACEPTE TODOS LOS TÉRMINOS ESTIPULADOS EN EL PRESENTE CONTRATO DE LICENCIA DE CÓDIGO BINARIO Y TÉRMINOS DE LICENCIA ADICIONALES (EN CONJUNTO DENOMINADOS "CONTRATO"). POR FAVOR, LEA EL CONTRATO DETENIDAMENTE. AL SELECCIONAR EL BOTÓN "ACCEPT LICENSE AGREEMENT" (O EQUIVALENTE) Y/O AL UTILIZAR EL SOFTWARE, ESTARÁ USTED CONFIRMANDO QUE HA LEÍDO LOS TÉRMINOS DEL CONTRATO Y ESTÁ DE ACUERDO CON ELLOS. SI ESTÁ USTED PRESTANDO SU CONFORMIDAD EN NOMBRE DE UNA EMPRESA U OTRA PERSONA JURÍDICA, ESTARÁ USTED AFIRMANDO QUE DISPONE DE LOS PODERES NECESARIOS PARA QUE ESA PERSONA JURÍDICA QUEDE VINCULADA POR LOS TÉRMINOS AQUÍ ESTABLECIDOS. SI NO DISPONE DE DICHOS PODERES, O SI NO ESTÁ USTED DISPUESTO A COMPROMETERSE CON TODOS LOS TÉRMINOS DEL PRESENTE CONTRATO, SELECCIONE EL BOTÓN "DECLINE LICENSE AGREEMENT" (O EQUIVALENTE) Y NO DEBERÁ UTILIZAR EL SOFTWARE EN ESTE PUESTO DE TRABAJO, NI EN NINGÚN OTRO SOPORTE EN EL QUE EL SOFTWARE ESTÉ CONTENIDO.

1. DEFINICIONES. "Software" significa los productos Java SE Platform en forma binaria que haya seleccionado para su descarga, instalación o uso y que hayan sido suministrados por Oracle o sus licenciantes autorizados, cualquier otro material en formato legible por equipos informáticos (incluso, pero sin limitarse a ello, bibliotecas, archivos fuente, archivos de cabecera y archivos de datos), cualquier actualización o corrección de errores suministrada por Oracle, y cualquier manual de usuario, guía de programación y otra documentación que le haya sido proporcionada por Oracle bajo el presente Contrato. "Servidores y computadoras de escritorio de propósito general" significa computadoras, incluyendo las de escritorio y portátiles, o servidores, usados para funciones de computación de carácter general bajo control del usuario final (como por ejemplo para correo electrónico, navegación de carácter general por Internet y herramientas ofimáticas, sin que pueda considerarse exhaustiva esta lista). El uso del Software en sistemas y soluciones que proporcionen funcionalidad dedicada (distinta de la mencionada anteriormente) o diseñados para su uso en aplicaciones software empotradas o específicas de la aplicación (por ejemplo, sin que esta lista pueda considerarse exhaustiva: software empotrado o integrado en sistemas de control industrial, teléfonos móviles inalámbricos, dispositivos portátiles inalámbricos, netbooks, quioscos, TV/STB, dispositivos Blu-Ray Disc, equipos de conmutación para control de red y telemáticos, impresoras y sistemas de gestión de almacenamiento y otros sistemas relacionados) están excluidos de esta definición y no les es aplicable este Contrato de licencia. "Programas" significa los applets y aplicaciones con tecnología Java concebidos para ejecutarse en la plataforma Java Platform Standard Edition (J2SE platform) o en Servidores y computadoras de escritorio de propósito general compatibles con Java. "Características comerciales" significa aquellas características identificadas en la Tabla 1-1 (Commercial Features In Java SE Product Editions) de la documentación software (Software documentation) disponible en http://www.oracle.com/technetwork/java/javase/documentation/index.html.

   "Archivo README" significa el archivo README del Software disponible en http://www.oracle.com/technetwork/java/javase/documentation/index.html.

2. LICENCIA DE USO. En virtud de los términos y condiciones dispuestos en el presente Contrato, incluidas, entre otras, las Restricciones de la Tecnología Java de los Términos de Licencia Adicionales, Oracle le concede, sin tarifa de licencia, una licencia limitada, no exclusiva e intransferible para la reproducción y el uso interno del Software completo y sin modificaciones con el único propósito de ejecutar Programas. LA LICENCIA CONCEDIDA EN ESTA SECCIÓN 2 NO ABARCA LAS CARACTERÍSTICAS COMERCIALES. SUS DERECHOS Y OBLIGACIONES EN RELACIÓN CON LAS CARACTERÍSTICAS COMERCIALES SON LOS ESTABLECIDOS LOS TÉRMINOS DE LICENCIA ADICIONALES, JUNTO CON LICENCIAS ADICIONALES PARA DESARROLLADORES Y/O EDITORES.

3. RESTRICCIONES. El software se encuentra protegido por derechos de autor (Copyright). Oracle y/o sus licenciantes mantienen la titularidad del Software, así como todos los derechos de propiedad intelectual asociados. Queda prohibido modificar, descompilar o utilizar técnicas de ingeniería inversa en el Software, a menos que se estipule lo contrario en la legislación aplicable. El licenciatario acepta que el Software se ha desarrollado para un uso general en diversas aplicaciones de tratamiento de la información; no se ha diseñado, ni está pensado, para ser utilizado en ninguna aplicación inherentemente peligrosa, incluyendo aquellas aplicaciones que puedan crear riesgo de daños personales. Si utiliza el Software en aplicaciones peligrosas, será responsabilidad suya tomar todas las necesarias medidas de seguridad, respaldo y redundancia para garantizar un uso seguro. Oracle renuncia a cualquier garantía explícita o implícita de adecuación del Software para dichos fines. El presente Contrato no otorga ningún derecho, título o propiedad sobre o respecto a las marcas comerciales o de servicio, logotipos o nombres comerciales de Oracle o de sus licenciantes. Las restricciones adicionales para las licencias de los desarrolladores y/o editores se estipulan en los Términos de Licencia Adicionales.

4. GARANTÍA LIMITADA. EL SOFTWARE SE SUMINISTRA "TAL CUAL", SIN GARANTÍA DE NINGÚN TIPO. ORACLE RENUNCIA ADEMÁS A TODAS LAS GARANTÍAS EXPLÍCITAS O IMPLÍCITAS, INCLUIDA CUALQUIER GARANTÍA IMPLÍCITA DE COMERCIABILIDAD, IDONEIDAD PARA UN PROPÓSITO DETERMINADO Y DE NO INFRACCIÓN DE OTROS DERECHOS.

5. LIMITACIÓN DE RESPONSABILIDAD. EN NINGÚN CASO ORACLE ASUMIRÁ RESPONSABILIDAD ALGUNA POR DAÑOS O PERJUICIOS ESPECIALES, INDIRECTOS, CONSECUENTES, INCIDENTALES, O PUNITIVOS, NI POR DAÑOS POR PÉRDIDA DE BENEFICIOS, INGRESOS, INFORMACIÓN O USO DE LA INFORMACIÓN, SUFRIDOS POR USTED O POR TERCERAS PARTES, YA SEA POR ACTUACIONES CONTRACTUALES O EXTRACONTRACTUALES, , INCLUSO EN AQUELLOS CASOS EN LOS QUE SE HAYA ADVERTIDO A ORACLE DE LA POSIBILIDAD DE QUE SE PRODUZCAN TALES DAÑOS. LA RESPONSABILIDAD TOTAL DE ORACLE POR DAÑOS, DE ACUERDO CON ESTO, NO EXCEDERÁ EN NINGÚN CASO DE MIL DÓLARES ($1.000).

6. RESCISIÓN. El presente Contrato se mantendrá vigente hasta que se produzca su rescisión. Puede usted rescindir el presente Contrato en cualquier oportunidad por medio de la destrucción de todas las copias del Software. Oracle podrá rescindir el presente Contrato en cualquier momento y sin notificación previa cuando usted no haya cumplido alguna de las cláusulas en él incluidas. Cualquiera de las partes podrá rescindir el presente Contrato de forma inmediata si el Software pasa a ser objeto, o en la opinión de cualquiera de las partes es probable que lo sea, de una reclamación por violación de los derechos de propiedad intelectual. Una vez rescindido el Contrato, deberá destruir todas las copias del Software.

7. NORMAS RELATIVAS A LA EXPORTACIÓN. El Software y toda la información técnica suministrada de conformidad con el presente Contrato se rigen por las leyes de control de exportaciones de Estados Unidos y otras normas relativas a la exportación o importación; puede encontrar información adicional en el sitio web Global Trade Compliance de Oracle (http://www.oracle.com/products/export). Usted se compromete a que el Software, así como cualquier producto directo obtenido a partir del mismo, no se exporten directa o indirectamente salvo de acuerdo con dichas leyes, y también se compromete a que no se utilicen para ningún propósito prohibido por dichas leyes, incluyendo entre otros la proliferación de armas nucleares, químicas o biológicas.

8. MARCAS COMERCIALES Y LOGOTIPOS. El licenciatario y Oracles acuerdan y reconocen que Oracle es propietaria de las marcas comerciales ORACLE y JAVA, así como de todas las marcas comerciales, marcas de servicio, logotipos y otras designaciones de marcas relacionadas con ORACLE y JAVA (en adelante denominadas “Marcas Oracle”). El licenciatario se compromete a cumplir los Requisitos de uso de las marcas comerciales de Oracle (Third Party Usage Guidelines for Oracle Trademarks) que encontrará actualmente en http://www.oracle.com/us/legal/third-party-trademarks/index.html. Todo uso que dé a las Marcas Oracle redundará en beneficio de Oracle.

9. DERECHOS RESTRINGIDOS DEL GOBIERNO DE ESTADOS UNIDOS. Si el Software con licencia es adquirido por o en nombre del Gobierno de Estados Unidos, o bien por uno de sus contratistas principales o subcontratistas (a cualquier escala), los derechos del Gobierno sobre el Software y la documentación adjunta quedarán limitados a lo establecido en el presente Contrato.

10. LEGISLACIÓN APLICABLE. Este contrato está sometido a la legislación de California. Oracle y usted acuerdan someter toda acción judicial que pudiera emprenderse en relación con este Contrato a la jurisdicción exclusiva de los tribunales de los condados de San Francisco o Santa Clara del estado de California.

11. INDEPENDENCIA DE LAS CLÁUSULAS CONTRACTUALES. La imposibilidad de cumplir alguna de las cláusulas del presente Contrato no afectará al resto del Contrato, que seguirá siendo válido sin dicha cláusula, a menos que la omisión de la misma pudiera perjudicar los propósitos de las partes, en cuyo caso se considerará rescindido este Contrato de forma inmediata.

12. TOTALIDAD DEL CONTRATO. A todos los efectos el presente Contrato se considerará como el contrato establecido entre usted y Oracle en relación con el objeto descrito. Por consiguiente, el presente Contrato invalida todo contacto, propuesta, manifestación o garantía que se haya efectuado entre las partes, anterior o actual, oral o escrito, y prevalecerá en todo momento sobre los términos adicionales o contradictorios de cualquier oferta, pedido, confirmación o comunicación relativos al objeto del Contrato y que pudieran llevar a cabo las partes durante el período de vigencia de éste. Las modificaciones efectuadas sobre el presente Contrato no resultarán en modo alguno vinculantes si no se presentan por escrito y firmadas por un representante autorizado de cada parte.

TÉRMINOS ADICIONALES DE LA LICENCIA

Estos Términos Adicionales de la Licencia amplían o modifican los términos del Contrato de Licencia de Código Binario. Los términos en mayúsculas que no se definan en los presentes Términos Adicionales mantendrán el mismo significado que se les ha atribuido en el Contrato de Licencia de Código Binario. Los presentes Términos Adicionales sustituyen cualquier término del Contrato de Licencia de Código Binario o de cualquier licencia contenida dentro del Software con los que sean contradictorios o incongruentes.

A. CARACTERÍSTICAS COMERCIALES. Usted no puede utilizar las Características Comerciales para ejecutar Programas, applets Java o aplicaciones en sus operaciones empresariales internas ni para ningún propósito comercial o de producción, ni para ningún otro propósito distinto de los establecidos en las Secciones B, D, D y E de estos Términos Adicionales. Si desea utilizar las Características Comerciales para cualquier propósito distinto de los autorizados por este Contrato, deberá usted obtener de Oracle una licencia independiente.

B. CONCESIÓN DE LICENCIA PARA USO INTERNO DEL SOFTWARE PARA DESARROLLO. Sujeta a los términos y condiciones dispuestos en el presente Contrato y las restricciones y excepciones estipuladas en el Archivo README aquí referenciado, incluidas, entre otras, las Restricciones de la Tecnología Java de los presentes Términos Adicionales, Oracle le

concede una licencia limitada, no exclusiva e intransferible, sin tarifa de licencia, para la reproducción interna y el uso interno del Software completo y sin modificaciones, con el propósito de diseñar, desarrollar y probar sus Programas.

C. LICENCIA PARA LA DISTRIBUCIÓN DEL SOFTWARE. Sujeta a los términos y condiciones del presente Contrato y las restricciones y excepciones estipuladas en el Archivo README, incluidas, aunque no exclusivamente, las Restricciones de la Tecnología Java de estos Términos Adicionales, Oracle le concede una licencia limitada, no exclusiva e intransferible, sin tarifa de licencia, para reproducir y distribuir el Software, siempre y cuando: i) distribuya el Software completo y sin modificar y únicamente integrado como parte de sus Programas y con el sólo propósito de ejecutarlos; ii) los Programas añadan una funcionalidad sustancial y primaria al Software; iii) no distribuya software adicional que pretenda sustituir cualquier componente del Software; iv) no elimine ni modifique las notificaciones ni los avisos de propiedad incluidos en el Software; v) distribuya el Software sólo mediante un contrato de licencia que: (a) sea una reproducción completa y sin modificaciones de este Contrato, o (b) proteja los intereses de Oracle de conformidad con los términos establecidos en el Contrato e incluya el aviso al que hace referencia la Sección G; y vi) acuerde defender e indemnizar a Oracle y a sus licenciantes por cualquier daño, costo, responsabilidad, acuerdo y/o gasto (incluidos honorarios de abogados) que se deriven de cualquier reclamación, litigio o acción de terceros como consecuencia del uso o distribución de cualquiera o de todos los Programas y/o el Software.

D. LICENCIA PARA REDISTRIBUIR LOS REDISTRIBUIBLES. Sujeta a los términos y condiciones del presente Contrato y las restricciones y excepciones estipuladas en el Archivo README, incluidas, aunque no exclusivamente, las Restricciones de la Tecnología Java de estos Términos Adicionales, Oracle le concede una licencia limitada, no exclusiva e intransferible, sin tarifa de licencia, para reproducir y distribuir aquellos archivos específicamente identificados como redistribuibles en el Archivo README ("Redistribuibles"), siempre y cuando: i) distribuya los Redistribuibles completos y sin modificar y únicamente integrados como parte de Programas; ii) los Programas añadan una funcionalidad sustancial y primaria a los Redistribuibles; iii) no distribuya software adicional que pretenda sustituir cualquier componente de los Redistribuibles (a menos que se especifique lo contrario en el Archivo README aplicable); iv) no elimine ni modifique las notificaciones ni los avisos de propiedad incluidos en los Redistribuibles; v) distribuya los Redistribuibles sólo mediante un contrato de licencia que: (a) sea una reproducción completa y sin modificaciones de este Contrato, o (b) proteja los intereses de Oracle de conformidad con los términos establecidos en el Contrato e incluya el aviso al que hace referencia la Sección G; y vi) acuerde defender e indemnizar a Oracle y a sus licenciantes por cualquier daño, costo, responsabilidad, acuerdo y/o gasto (incluidos honorarios de abogados) que se deriven de cualquier reclamación, litigio o acción de terceros como consecuencia del uso o distribución de cualquiera o de todos los Programas y/o el Software.

E. DISTRIBUCIÓN POR EDITORES. Esta sección hace referencia a su distribución del Software JavaTM SE Development Kit con su revista o libro impreso (ya que dichos términos se usan comúnmente en el sector) relacionado con la tecnología Java ("Publicación"). Sujeta a las restricciones y obligaciones del presente Contrato, y condicionada a su cumplimiento de las mismas, Oracle le concede el derecho limitado, no exclusivo e intransferible para reproducir copias completas y no modificadas del Software en soporte electrónico (los "Soportes") con el solo propósito de incluirlos en sus Publicaciones y distribuirlos con ellas, sujeta a los términos siguientes: (i) no puede distribuir el Software por sí solo, sino que debe ser distribuido con sus Publicaciones; (ii) es usted responsable de descargar el Software del sitio web de Oracle aplicable; (iii) debe denominar al Software de la forma siguiente: JavaTM SE Development Kit; (iv) el Software debe ser reproducido en su totalidad y sin ninguna modificación (incluyendo todo lo referido a los avisos de propiedad) y debe ser distribuido junto con su Publicación sujeto a un contrato de licencia que sea una reproducción completa y sin modificaciones de este Contrato; (v) la etiqueta del Soporte deberá incluir la siguiente información: Copyright 2011, Oracle America, Inc. Todos los derechos reservados. El uso está sujeto a los términos de licencia. ORACLE y JAVA son marcas comerciales y todas las marcas comerciales, marcas de servicio, logotipos y otras designaciones de marcas relacionadas con ORACLE y JAVA son marcas comerciales o marcas registradas de Oracle en los EE.UU. y otros países. Esta información debe colocarse en la etiqueta del Soporte de modo tal que sólo sea aplicable al Software de Oracle; (vi) debe identificar claramente el Software como un producto de Oracle en el contenedor del Soporte o en la etiqueta del Soporte, y no pude afirmar o sugerir que Oracle es responsable de ningún software de terceras fuentes contenido en el Soporte; (vii) no puede incluir en el Soporte ningún software de terceras fuentes que pretenda sustituir o reemplazar el Software; (viii) usted acuerda defender e indemnizar a Oracle y a sus licenciantes por cualquier daño, costo, responsabilidad, acuerdo y/o gasto (incluidos honorarios de abogados) que se deriven de cualquier reclamación, litigio o acción de terceros como consecuencia del uso o distribución del Software y/o la Publicación; y (ix) deberá notificar por escrito a Oracle de cada Publicación, debiendo dicha notificación incluir la siguiente información: (1) título de la Publicación, (2) autor(es), (3) fecha de la Publicación y (4) números de ISBN o ISSN. Dicha notificación deberá ser enviada a Oracle America, Inc., 500 Oracle Parkway, Redwood Shores, California 94065 U.S.A, Attention: General Counsel.

F. RESTRICCIONES DE LA TECNOLOGÍA JAVA. Usted se compromete a no crear, modificar ni alterar el comportamiento, y a no autorizar a sus licenciatarios para crear, modificar ni alterar el comportamiento, de las clases, interfaces ni subpaquetes que en cualquier modo se identifiquen como "java", "javax", "sun", "oracle" o similares, según especifique Oracle en cualquier designación de convenio de denominación.

G. AVISO SOBRE CARACTERÍSTICAS COMERCIALES. Con el propósito de cumplir con las Secciones C.(v)(b) y D.(v)(b) de estos Términos Adicionales, su acuerdo de licencia debe incluir el siguiente aviso, teniendo que estar el aviso mostrado de manera que cualquiera que utilice el Software lo vea::

El uso de las Características Comerciales para cualquier propósito comercial o de producción requiere obtener de Oracle una licencia independiente. "Características comerciales" significa aquellas características identificadas en la Tabla 1-1 (Commercial Features In Java SE Product Editions) de la documentación software (Software documentation) disponible en http://www.oracle.com/technetwork/java/javase/documentation/index.html.

H. CÓDIGO FUENTE. El Software puede contener código fuente que, a menos que se otorgue una licencia expresa para otros fines, se proporciona únicamente con fines de referencia en virtud de los términos del presente Contrato. El código fuente no podrá redistribuirse a menos que así se estipule explícitamente en el presente Contrato.

I. CÓDIGO DE TERCEROS. En el archivo THIRDPARTYLICENSEREADME disponibles en http://www.oracle.com/technetwork/java/javase/documentation/index.html se exponen avisos adicionales de derechos de autor y términos de licencia aplicables a partes del Software. Además de cualsquiera de los términos y condiciones de cualquier licencia de código abierto/freeware de terceras fuentes identificadas en el archivo THIRDPARTYLICENSEREADME, las disposiciones de garantía limitada y limitación de responsabilidad comprendidas en los párrafos 5 y 6 del Contrato de Licencia de Código Binario se aplicará a la totalidad del Software contenido en esta distribución.

J. RESCISIÓN POR INFRACCIÓN. Cualquiera de las partes podrá rescindir el presente Contrato de forma inmediata si el Software pasa a ser objeto, o en la opinión de cualquiera de las partes es probable que lo sea, de una reclamación por violación de los derechos de propiedad intelectual.

K. INSTALACIÓN Y AUTO-ACTUALIZACIÓN. Los procesos de instalación y auto-actualización del Software transmiten una limitada cantidad de datos a Oracle (o a su proveedor de servicios) acerca de esos procesos específicos, con el fin de ayudar a Oracle a comprenderlos y optimizarlos. Oracle no asocia esos datos con ninguna información personal susceptible de ser identificada. Puede encontrar más información acerca de los datos que Oracle recopila como resultado de su descarga del Software en http://www.oracle.com/technetwork/java/javase/documentation/index.html.

Si tuviera alguna duda, escriba a: Oracle America, Inc., 500 Oracle Parkway, Redwood Shores, California 94065 USA.

Última actualización: 17 de mayo de 2011